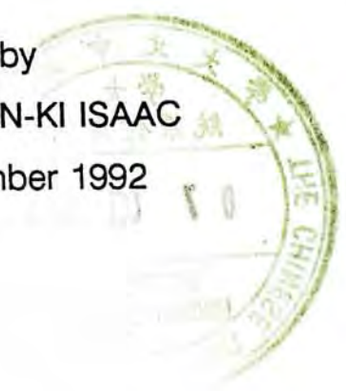
Nontermination Debugging of Prolog Programs

A Thesis
presented to the Department of Computer Science
of the Chinese University of Hong Kong
in partial fulfillment of the requirements
for the Degree of Master of Philosophy

by
LAM, HIN-KI ISAAC
December 1992



ACKNOWLEDGEMENTS

I would like to acknowledge my supervisor Dr. M.C. Lee for his supervision and critiques of my work. His comments are crucial in developing the methods presented in the present thesis. I would also like to acknowledge Dr. L.M. Liu for his valuable suggestions to my work while he was still working at the Chinese University of Hong Kong. May God bless him on his further study in theology. Also, I would like to thank my wife Angel M.Y. Lin for giving me support and encouragement throughout my M.Phil. program. Lastly, I would like to give my deepest gratitude to God Almighty for giving me wisdom and health to accomplish my work.

Abstract

Though Prolog is supposed to be a declarative programming language, nontermination is a common phenomenon in recursive procedures written by the novice Prolog programmers. The detection of nontermination errors inherent in Prolog programs has been investigated by various contemporary research workers. Yet no sound nontermination diagnosis schemes have been devised. This thesis investigates the nontermination issue in pure Prolog programs and develops some algorithms based on certain compile-time techniques for nontermination detection. The compile-time techniques already devised include static program structure analysis, parameter analysis, and data analysis, which correspond to the three essential steps for nontermination detection.

To diagnose any nontermination problem in a given program, static program structure analysis is first performed to identify all the recursive rules. Then parameter analysis is carried out on each recursive rule to construct cyclic parameter links. The absence of any cyclic parameter link implies that there is a nontermination error in the recursive procedure. If one or more cyclic parameter links are found, data analysis has to be performed on each of the parameter links in order to construct a set of connected data-link lists. A connected data-link list corresponds to one possible sequence of parameter values to be passed over the cyclic parameter link through some parameter cycles of recursion. If for each cyclic parameter link, there is at least one cyclically connected data-link list, it can be concluded that the recursive procedure will not terminate.

Despite some limitations, the present algorithms can handle many of the common recursive definitions in pure Prolog. Moreover, parameter analysis and data analysis have been shown to be powerful tools for detecting nontermination. These techniques provide a sound foundation on which further research can be done in order to enhance the nontermination detection capability of the present algorithms.

TABLE OF CONTENTS

CHAPTER 1— Introduction

1.1 The Problem

In *Algorithmic Program Debugging*, Shapiro presented a debugging scheme for logic programs. Based on the procedural semantics of logic programs, he distinguished three types of semantic errors in logic programs: (i) termination with incorrect output, (ii) termination with missing output, and (iii) nontermination [14]. Shapiro also provides algorithms to diagnose erroneous procedures. His algorithms are all based on a dynamic tracing technique.

However, any debugging algorithm based a dynamic tracing technique can encounter a problem in detecting nontermination: the simulation of a nonterminating program may not be able to terminate, and consequently the nontermination detection process may become nonterminating. In order to prevent from getting into a nontermination situation, Shapiro's nontermination diagnosis algorithm always requires its users to supply a safeguard upperbound for any given program. Once the upperbound is reached, the algorithm stops tracing the program any further no matter if a conclusion, concerning whether the program terminates, can be made. In this situation, it may not be able to determine if the program will terminate. Shapiro was aware of this weakness and admitted that his nontermination diagnosis algorithm "may fail to detect an error in a program that exhausted a resource, and in such a case it is up to the programmer to decide which ... action to take" [15].

Shapiro's nontermination diagnosis algorithm has another shortcoming since it cannot locate precisely what causes a nontermination error. Unlike the algorithm used for detecting termination with incorrect output or termination with missing output, which

can identify a particular procedure responsible for the error, the nontermination diagnosis algorithm can only return the sequence of procedure calls which repeats itself indefinitely if any nontermination can be detected at all. Evidently, the other problem is that since the nontermination diagnosis algorithm cannot identify the exact location of an error, it is up to the user to find out why nontermination occurs in a particular sequence of calls. (Shapiro's nontermination diagnosis algorithm will be examined more closely in Section 1.2 below; further details can be found in [14] and [13].)

Nontermination is a general phenomenon and a serious problem in Prolog programming. As the definition of a Prolog program is supposed to be based on a declarative programming paradigm, a novice programmer may not be aware of any nontermination errors inherent in his Prolog programs. Therefore, there is indeed a need to develop nontermination detection algorithms. In view of the limitations of Shapiro's nontermination diagnosis algorithm, this thesis attempts to explore the possibility of developing more powerful diagnosis methods. Due to the time constraint, the scope of the present investigation has been confined to the problem of nontermination in **pure Prolog** programs. In particular, the algorithm to be developed should satisfy the following criteria:

a) It should not become nonterminating while diagnosing nontermination errors in a given Prolog program.

b) It should not require the user to provide information such as stack depth like what is required by Shapiro's algorithm before detecting any potential nontermination error present in a given program. This is important since the programmer may not be able to provide the accurate information.

c) It should be able to locate from a given program the procedure causing nontermination. In addition, it should also be able to identify accurately which one of the rules defining the procedure is responsible for the nontermination. Better still, if it can locate which part of the rule, which causes the error. It would be ideal if the information provided by the algorithm at the end of the diagnosis can help the user to realize why nontermination occurs in his program.

1.2 Related Works

Shapiro's approach to the diagnosis of nontermination is based on a concept he called *well-founded ordering*. A set of elements is considered to be in a well-founded ordering if

(1) the set is not infinite; and

(2) any pair of these elements are in a binary relation which is **transitive, asymmetric and irreflexive.**

Shapiro considers terminating programs as those where the computation can only generate procedure calls which are in a well-founded ordering. In other words, the well-founded ordering of procedure calls is considered to be the characteristic for the termination of a Prolog program. Therefore, his nontermination detection method relates to detecting any procedure call sequence that violates the ordering.

However, Shapiro does not suggest how to implement an exhaustive search for violation of a well-founded ordering in a sequence of procedure calls. Instead, in *Algorithmic Program Debugging*, Shapiro suggests an algorithm that searches for a looping segment in a procedure call sequence. A looping segment, or a loop, appears in a procedure call sequence when a procedure is repeatedly called with the same input data or parameter values. Since two procedure calls are in a well-founded ordering only if they are in an asymmetric and irreflexive relation, the well-founded ordering is obviously violated in a procedure call sequence if certain procedures are called repeatedly with the same parameter values.

Evidently, Shapiro's nontermination diagnosis algorithm is based on the detection of a looping segment in a particular procedure calls sequence. He makes use of a stack to keep track of all the procedure calls. The height of the stack is supplied by the user who is assumed to have knowledge about the intended behavior of the program being tested. In his algorithm for nontermination detection, the procedure being tested is simulated with a particular input and the procedures called during the evaluation are put on the stack. The process then goes on until the simulation is completed by itself

or the stack depth exceeds a given upperbound. In the former case, the implication is that the procedure is free from nontermination with a particular input. However, this does not imply that the program being tested is free from nontermination under all circumstances. In the latter case, the simulation is aborted and the procedure calls in the stack are examined. If a looping segment is found, the algorithm returns the segment. Otherwise, a message is returned to the user to indicate that it is not certain whether the test program has a nontermination problem. Since Shapiro's algorithm for nontermination detection needs to simulate the evaluation of the procedure being tested, it can be classified as a kind of run-time or dynamic tracing technique.

One of the most obvious shortcomings of Shapiro's algorithm concerns the way the stack is used. Since the simulation is aborted once the stack height is exceeded, determination of the stack height becomes very important for a successful search for the looping segment. If it is too small, the simulation will be aborted without yielding any useful information. Consequently, when a simulation is aborted without returning a looping segment, we cannot be sure whether the stack height is too small, or the procedure being tested is nonterminating. On the other hand, if the stack height is too large, it becomes very time-consuming to run the test. Since the height of the stack is supplied by the user, it is assumed that the user has a very clear knowledge of the intended behavior of his program. However, such an assumption may not always be valid.

Gelder's *Tortoise-and-Hare* technique represents a significant improvement on Shapiro's approach [6]. Gelder's method no longer requires the user to provide the stack height. But two pointers, the *hare* and the *tortoise*, are required to point to different points of the stack. When simulating the execution of a procedure, the stack building steps are alternately labelled as *hops* and *walks*, starting with a *hop*. In the hopping steps, only the *hare* pointer can move up the stack by one level while in the walking steps both pointers move up the stack. The result is that the *hare* always stays at the top of the stack while the *tortoise* always points at about the middle of the stack. After each step, the elements pointed by the two pointers are compared. If they are *essentially the same*, a looping segment is found; otherwise, the process goes on until a

looping segment is found or is terminated by the user. Two procedure calls are *essentially the same* if they differ only in variable names. For example, two Prolog goals *a(1,X)* and *a(1,Y)* are *essentially the same* because the variables *X* and *Y* can always be unified. Since the variables in a procedure call is renamed by Prolog during the evaluation, Gelder specifically introduces the concept of *essentially the same* to avoid overlooking certain possible looping segments in Prolog programs. The *Tortoise-and-Hare* technique works because the two pointers are always kept apart with a distance of half of the procedure calls sequence. Therefore, the distance between them increases as the height of the stack grows. If the procedure calls are in an infinite loop with a looping segment of length ***n***, the distance between the two pointers will eventually grow to a multiple of ***n*** and catch the looping segment. Similar to Shapiro's algorithm, Gelder's *Tortoise-and-Hare* technique is also a kind of run-time tracing technique.

Apart from the attempt to develop diagnostic methods for nontermination in Prolog programs, there have been efforts to tackle the weakness of the conventional Prolog execution model in coping with certain special loops. Covington [1] notices that expressing transitive relations, symmetrical relations or biconditionals in Prolog programs can lead to infinite loops. Although the aim of Covington's work is to modify Prolog's implementation in order to enhance the power of Prolog to express transitive and symmetrical relations and biconditionals, his works [1,2] relate to nontermination detection in two respects: first, the algorithm suggested in his works can be used for diagnosing nontermination caused by transitive and symmetrical relations and biconditionals; second, his works show that some infinite loops are caused by inappropriate Prolog program structures. Although the first aspect of his work can only be incorporated into a run-time tracing algorithm, the second aspect implies that at least some nontermination errors can be detected by a compile-time analytical approach. Furthermore, when Nute tries to tackle a similar problem [8], he points out that a loop will only occur in a Prolog program with some recursive definitions. Poole and Goebel [10], on the other hand, suggest that the elimination of loops in Prolog programs can be better performed by modifying the program instead of using the methods suggested by Covington and Nute (see [1], [2] and [8]). Kowalski also points out that an inappropriate recursive definition can result in nontermination [7]. He notices that

there are two types of infinite loops [7]. Apart from the infinite loops discussed by Shapiro and Gelder, there are infinite loops caused by divergent recursive calls. A divergent recursive call can generate an infinite sequence of procedure calls in which there is neither any exact looping segment nor any elements which are *essentially the same*. Although he does not directly suggest any algorithm to handle the inappropriate recursive definition, he suggests that one can identify the infinite recursive calls by analyzing the change of the argument values of the recursive definition during its evaluation. Therefore, his work provides an approach to detect infinite loops that cannot be detected by Shapiro's and Gelder's run-time tracing technique. Although his approach also suggests a run-time tracing technique, it shows that analysis on the arguments of a recursive definition can be crucial in detecting nontermination in certain Prolog programs.

On the other hand, some researchers have tried to explore the compile-time analytical approach to nontermination detection in Prolog programs. In De Schreye et al's work [12], a directed, weighted graphs technique is employed to detect nonterminating queries for the recursive definitions of a restricted class: recursive definitions with left-recursive rules in the form "P(...) :-P(...)". In [11], De Schreye et al prove that there is a necessary and sufficient condition for the existence of a query that is nonterminating in the absence of occur check. By representing the evaluation generated by a query with a rational tree, one can adopt a mathematical approach to analyze the evaluation of the recursive definition with left-recursive rules. This analysis shows that it is possible to associate a weighted, directed graph to a recursive definition with left-recursive rules if and only if the recursive definition can admit a nonterminating query. Although their nontermination detection algorithm has limited computational complexity at compile time and can be easily implemented, it suffers from a great disadvantage: it has very limited scope of applicability. There are also other nontermination detecting techniques based on the compile-time approach [5,16]. Nontermination is detected by a combination of global analysis and methods which prove that the length of certain data structures becomes increasingly shorter during the evaluation of the recursive definition.

A more general solution is explored under the mathematical approach. In [3], Baudinet tries to develop a method to prove the termination properties of Prolog programs. As a system of functional equations, a Prolog program actually maps a goal to the sequence of answer substitutions that can be generated when the goal is supplied as a query for the program. Such a sequence can be either finite or infinite depending on whether a finite or infinite number of answers is produced. In Baudinet's method, how to translate a Prolog program into some appropriate semantics equations is a central concern. By transforming a Prolog program to a system of functional equations of which the least fixpoint is the meaning of the program, he shows that termination or nontermination properties can be proved by reasoning with these functional equations and using fixpoint induction or structural induction. Usually, structural induction is sufficient to prove the universal termination of a program for the type of goals that have finite and proper answer sequences. When a program loops, fixpoint induction is needed. Since the properties can only be obtained through reasoning the program equations through fixpoint or structural induction, the implementation of Baudinet's method requires a run-time tracing approach. However, in general, the mathematical approach employed in this method indicates an analytical approach to nontermination detection can be fruitful.

Plümer also uses a mathematical approach to provide a termination proof for Prolog programs [9]. However, his focus is on the recursive procedure with recursive data structures. In fact, his work is an attempt to overcome the restrictions of the technique suggested in [16]. In order to handle the case of certain complicated recursive data structures, the notion of *linear predicate inequalities* is introduced in his work. The presence of linear predicate inequality is a termination proof for a recursive procedure with recursive data structures. In Plümer's method, linear predicate inequalities are derived using the technique of AND/OR dataflow graph. In other words, he provides a compile-time analytical approach to nontermination detection for a special class of Prolog programs: programs with recursive procedures that have recursive data structures. More importantly, his work shows the possibility of incorporating graph techniques in the Prolog nontermination detecting methods.

1.3 Contribution of the Thesis

The major contributions of the thesis include the following:

i) It develops a compile-time approach based on static program structure analysis to detect nontermination in pure Prolog programs. As this approach does not require the simulation of the execution of a program being examined, therefore, the diagnosis algorithm would not become nonterminating.

ii) The methodology developed in this thesis can detect nontermination in different types of Prolog programs, unlike many of those devised by contemporary workers, which can only handle Prolog programs having some special structures.

iii) The algorithm based on static structure analysis can locate relatively precisely which part of a given program responsible for causing nontermination.

iv) The algorithms developed in this thesis can be used to generate useful information which may help the user to understand why his/her program does not terminate.

v) The parameter and data analysis techniques developed in this thesis provides a theoretical framework for nontermination diagnosis in pure Prolog programs. The techniques can also be applied to full Prolog programs with some restrictions.

vi) The successful development of the parameter analysis and the data analysis technique for nontermination detection provides insight into the cause of nontermination in a recursive Prolog program. With such a knowledge, a Prolog programmer can more easily prevent writing nonterminating recursive programs.

1.4 Outline of the Thesis

In Chapter 2, the basic concepts of logic programming and the Prolog execution model are reviewed. Then we investigate how nontermination can occur in a pure Prolog program; the concepts of exit condition and of exit-reaching process in the context of a recursive procedure in a conventional programming language are introduced. We show how such concepts can be applied to a recursive Prolog definition. The presence of an exit-reaching process implies that there must be one or more variables in the exit condition; and the values of the variables must be modified by the

exit-reaching process during recursion. In a Prolog recursive definition, a variable of the exit condition must also be related to a parameter, an exit-reaching process should a parameter modifying process. At the end of this chapter, we briefly mention the possibility of developing a preliminary test for nontermination error in a Prolog program based on the analysis of parameters.

In Chapter 3, we develop parameter analysis algorithms. In Section 3.1, we show what a parameter link is. Then we explore how a parameter link can be formed in different ways. In Section 3.2, we explain how parameter links can be connected to form a cyclic parameter link. These two sections also explain how a cyclic parameter link can be related to the parameter modifying process. In Section 3.3, we introduce graphical notations for representing parameter links and cyclic parameter links. Then algorithms for constructing parameter links and cyclic parameter links are presented.

In Chapter 4 we develop algorithms for data analysis. In Section 4.1, we explain what a data link is and how it can be formed for a cyclic parameter link of a recursive definition. In Section 4.2, we show the difference between pure Prolog and general Prolog, and illustrate how such difference can affect the method for data analysis. In Section 4.3, we show the relationship between data links and nontermination. In Section 4.4, we illustrate how data links can be employed to detect nontermination. It is also shown that data links can be connected to form connected data-link lists which can represent the data transfer through the cyclic parameter link. Consequently, nontermination can be detected by examining the connected data-link lists. In Section 4.5, the construction of data links and connected data-link lists for special parameters are considered. Finally, the algorithms for data analysis are presented in Section 4.6.

In Chapter 5, two special cases are considered to enhance the power of data analysis. In Section 5.1, the case of interdependent cyclic parameter links is examined. We show how interdependent cyclic parameter links can arise from cyclic parameter links sharing certain common parameters or common subgoals. Then we present how the interdependency can cause the method of data analysis mentioned in Chapter 4 to give incorrect conclusions. We also discuss what adjustments of the foregoing data

analysis method need to be made in order to handle recursive definitions with interdependent cyclic parameter links. Concerning the case of cyclic parameter links established through special parameters, data analysis may be inadequate for detecting nontermination in a special situation. In Section 5.2, we investigate why data analysis may fail in some special situations.

In Chapter 6, the results of the study are presented. Then the limitations of the our algorithms and the future research are discussed. Finally, a conclusion is made.

CHAPTER 2 —Nontermination and Recursive Definitions

In this chapter, we shall first present an overview of the Prolog execution model. Through a discussion of the execution model, we shall see that Prolog is different from conventional programming languages in one important respect: nontermination in Prolog programs occurs **only in the form of recursive definitions.** A Prolog program will terminate properly only if all of its recursive definitions include certain termination requirements. In general, the requirements for terminating a recursive definition consist of **exit conditions** and **an exit-condition reaching process** [7]. Nontermination in Prolog programs occurs when any one of these termination requirements is absent from the recursive definitions in a Prolog program. In the rest of this chapter, we shall analyze what exactly constitutes the exit condition and the exit-reaching process in Prolog. This analysis shows two important aspects of the termination requirements. Firstly, it shows that either the exit condition or the exit-condition reaching process is formed by the parameters and/or the subgoals in the recursive rule only. Secondly, it shows that the exit condition and the exit-condition reaching process are in an interdependent relationship; the existence of one of them implies the existence of the other. These findings indicate the possibilities of developing a nontermination detection system based on analyzing parameters.

2.1 Prolog Execution Model

In this section, an overview of Prolog and its execution model is given to provide a background to the reader. However, the reader who needs more details about Prolog can refer to [13] and [17]. Through a discussion of the execution model, we shall

illustrate how nontermination arises in a Prolog program and how nontermination relates to recursive definitions.

A pure Prolog program, as a logic program, is composed of a bundle of **Horn clauses**, or **clauses**. There are three types of clauses: **facts**, **rules** and **queries**.

The simplest kind of clause is **fact**. A fact states a specific relationship that holds between certain objects. It has a format as

$$f(a_1, a_2, ..., a_m). \qquad \text{where } m >= 0$$

f is usually known as the predicate name while $a_1, ..., a_m$ are known as arguments. The predicate name f represents the relationship while the arguments $a_1, ..., a_m$ represent the objects. In each fact, the arguments can be either some specific values or some **variables**. A specific value is also known as an **atom**. The predicate name must be an atom. If a variable is used as the arguments instead of an atom, it can be instantiated to whatever objects used, even another variable. A finite set of facts forms the simplest form of logic program. Atoms and variables are also collectively known as **terms**. Moreover, a structure $f(t_1, t_2, ..., t_n)$ is also a term if f is an atom and $t_1, t_2, ..., t_n$ are either atoms, variables or structures. A term is known as a **compound term** if some structures appear as its arguments. For example, $f(1,2,3)$ is a simple terms but $f(X, 1, g(D, 2))$ is a compound term. A term not consisting of any variable is considered as a **ground term**.

Rules express a conditional relation between some existing relationships. It has a general format as:

$$g :- s_1, s_2, ..., s_n. \qquad \text{where } n >= 0$$

g is the **head** of the rule and s_i's are the **body**. They are all **goals**. Goals in the body are known as **subgoals**. Each goal has zero, one or more arguments. Actually a fact can be viewed as a special case of rules with $n = 0$. Each subgoal in the body of a rule must be defined either as facts or rules somewhere else in the program. The symbol :- is used to denote the implication relation existing between the head and the body, whereas the head is the conclusion of the preconditions specified in the body. To determine the truth value of the head or the values of the arguments used in the head, each subgoal

must first be evaluated to determine the atomic values for each of its variables. A finite set of rules and facts constitutes a program. Moreover, a set of rules and facts where the rule heads and the facts have the same relation name form a **procedure.**

A procedure is known as a **direct recursive definition** if the head of one of its rules also appears as a subgoal in the body (they can have different parameters). This rule is referred to as a recursive rule and the subgoal is known as a **recursive subgoal.** The head of this rule is known as a **recursive rule head.** Moreover, if there exist several rules from different procedures such that each of these rules has its head also appearing as a subgoal in another rule, these rules form an **indirect recursive definition.** These subgoals are in fact indirect recursive subgoals. If we start from any one of these rules to replace its recursive subgoal with the body of the corresponding rule, we can eventually produce a rule having its head as a subgoal in the body, just as the case of direct recursive definitions. Each rule forming this recursive definition (even though they are from different procedures) is also referred to as a recursive rule.

Queries are the clauses that retrieve information from a logic program. They have the following general format:

?- g_1, g_2, ..., g_n. where $n >= 1$

Each subgoal in a query, g_i, has the general format of a goal. If all subgoals in a query are ground terms, we can interpret the query as a question of whether all relations represented by the subgoals can hold at the same time among the objects specified as the arguments of these subgoals. Usually, uninstantiated variables appear in a query. In this case, the query can be viewed as a question for finding the unknowns represented by the variables. For $n > 1$, all subgoals are solved one by one, and a query is considered to be solved only after every subgoal has been solved.

Sometimes a fact appears as a compound term to allow the data to be stored in a more organized style. For example, if we want to store some information about an event, it is more meaningful to store the data as:

event(place(Where), time(10,25,am), date(12,6,89)).

than

event(Where, 10, 25, am,
12, 6, 89).

These non-atomic terms that are used as the arguments of facts or goals (e.g., in this case, *place(Where)*, *time(10,25,am)*) are known as **structured data**. On the other hand, Prolog provides another method to store complicated data. By enclosing all related data within a pair of square brackets, "[ ]", Prolog can handle them together as one object. Such a structure is known as a **list**. For example,

[event [place, Where],
[time, 10, 25, am],
[date, 12, 6, 89]]

INPUT : two terms, T_1 and T_2, to be unified

OUTPUT : the most general unifier, **mgu**, or **failure**

```
Initialize the mgu to be empty

Push "T1 = T2" on a stack

WHILE the stack is not empty
  DO { Pop "X = Y' from the stack

       IF X and Y are NOT identical atoms
        THEN
          IF X is a variable
           THEN { Substitute Y for X in the stack
                  Add "X = Y" to mgu
                }
          ELSE IF Y is a variable
           THEN { Substitute X for Y in the stack
                  Add "Y = X" to mgu
                }
          ELSE IF X and Y are variables
           THEN Rename all X and Y in the
                stack to the same name
          ELSE IF X is f(X1, ..., Xn) and
             Y is f(Y1, ..., Yn) where n > 0
           THEN Push "Xi = Yi", i = 1, ..., n,
                on the stack
           ELSE Exit and return failure
     }

Return the mgu
```

Figure 2.1 The unification algorithm

is a list of lists containing more or less the same information encoded in the structured data above.

In order to answer a query, the clauses defined in the program are used to satisfy each subgoal in the query. Since Prolog is the realization of logic programming in a sequential machine, it follows a specific sequence to satisfy the subgoals in a query. The leftmost subgoal is first selected. If it can be satisfied, other subgoals are selected one

by one in a **left-to-right** sequence. However, if one of the subgoals fails to be satisfied, the process will go back to the left subgoal of the failing one and attempts to re-satisfy it with an alternative clause. This process is known as **backtracking**. If the left subgoal does not have an alternative clause, backtracking will continue to the next left subgoal until one of those subgoals on the left can be re-satisfied with another clause, or it reaches beyond the leftmost subgoal. In the latter case, the query results in a failure. On the other hand, after all subgoals in a query are satisfied, a solution can be obtained for the query. However, a Prolog user can initiate backtracking himself/herself after a solution is found. During this backtracking process, other possible clauses in the program are tried so that alternative solutions can be found. Satisfying a subgoal greatly depends on matching this subgoal to other terms in the program. This matching process is known as **unification**. Its algorithm is shown in Figure 2.1.

In the algorithm, if a fact is used for unification with the subgoal, the subgoal can be satisfied immediately or the unification fails immediately. However, Prolog also attempts to use a rule to satisfy a subgoal. In this case, the subgoal is to be unified with the head of the chosen rule. If it succeeds, the unifier obtained in the unification process is applied to the body of the chosen rule. Then the subgoals in the body of this chosen rule are used to replace the original subgoal in the query to form a transformed query. This replacement process continues by following a **depth-first** strategy. In other words, the leftmost subgoal in the body of the chosen rule (it does not need to be the leftmost subgoal of the transformed query) will be replaced with a corresponding rule body again recursively until the leftmost subgoal can be unified with only a fact. Then the replacing process is applied to the second left subgoal. This process stops only when all subgoals in the transformed query can unify with some facts.

2.2 Nontermination

As pure Prolog is a declarative language, it does not provide any control constructs. A pure Prolog program describes only the logic component of algorithms

and the Prolog execution model itself will take care of the control component [7]. Therefore, unlike conventional programming languages, no looping construct exists in Prolog. A programmer can define iterations in a pure Prolog program by only two techniques:

(1) backtracking mechanism and

(2) recursive definitions [7].

Using backtracking to generate iteration is a technique unique to Prolog. To see how an iteration can be generated by a backtracking mechanism, let us consider the case of supplying a query *?- path(X,Y)* to Program (a) in Figure 2.2. After Prolog finds the first solution $X = a$, $Y = b$, the backtracking mechanism in Prolog will search the procedure *path* again to find all the other possible solutions. As a result, a total of five solutions can be found since there are five facts in the procedure *path* that can satisfy the query. From the operational point of view, the procedure call *path(X,Y)* is repeated five times and we can consider it as a five-time iteration.

However, in contrast to the iterations generated by the looping constructs in conventional programming languages, the iteration generated by backtracking in Prolog has two properties:

(1) it does not need any explicit condition for its termination, and

(2) it can always terminate if no recursive definition is involved.

These properties actually result from the fact that the possible search space in a Prolog program has only a limited size if no recursive definition is present. Once the whole search space in a Prolog program is completed, backtracking stops so that no explicit condition is needed to stop the iteration. Moreover, if the possible search space for a program has only a finite size, iteration generated by backtracking must stop after such a finite space has been completely searched; thus it can always terminate.

But how can we be sure that the possible search space for a "recursive definition free" Prolog program is always finite? The answer lies in how Prolog searches the program during the backtracking process. As has been discussed in Section 2.1, Prolog

tries to find some terms from the program to unify with each subgoal in the query from the leftmost subgoal to the rightmost subgoal. In this process, a suitable term is found by searching the corresponding procedure that has the same predicate name as this particular subgoal. If the head of a clause in the corresponding procedure can be unified with the subgoal in the query, this process will continue on the next immediate right subgoal. On the other hand, if it is the rightmost subgoal in the query, backtracking will resume the search on the rest of those not yet searched clauses in the procedure after a solution is found. However, if the unification process of this particular subgoal fails, backtracking occurs and renders the procedure corresponding to the immediate left subgoal to be searched again. The search will resume at the clause next to the previously searched one.

Therefore, the search space must be finite if all the subgoals in a certain query are all defined by **procedures that consist of facts only**. While a subgoal is defined by a procedure consisting of facts only, the search space for this subgoal is finite. As the search space for each subgoal of a rule is finite, the search space that can be generated by the backtracking process among these subgoals is also finite. It is obvious that the number of iterations generated cannot be greater than the product of the number of facts in each procedure defining the corresponding subgoal in the query. If we supply the query *?- path(X,Y), path(Y,Z)* to Program (a) in Figure 2.2, the upperbound for the number of iterations is 25. Actually, the number of iterations can be generated is also 25 although many facts cannot succeed in the unification process.

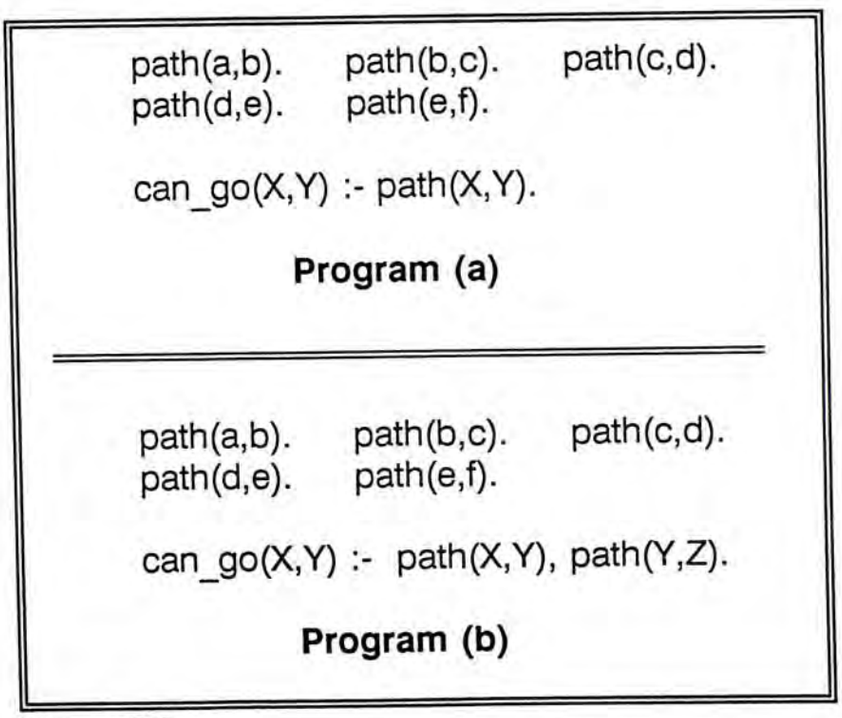


Figure 2.2

In general, there also exists an upperbound for the search space if **the procedures defining certain subgoals in a query is composed of not only facts but also some non-recursive rules**. The upperbound for the number of iterations generated can be calculated as the product of the upperbound for the number of iterations generated by each subgoal. For a subgoal with a procedure of all facts, the upperbound is equal to the number of clauses in the procedure. For a subgoal with a procedure consisting of facts and non-recursive rules, the upperbound for the number of iterations generated by this subgoal is equal to the number of facts plus the upperbound for the number of iterations generated by every rule. And the upperbound of a rule is just equal to the product of the upperbounds of all subgoals in this rule. For example, in Program (b) in Figure 2.2, if we supply the query *?- path(X,Y), can_go(Y,Z)* , the upperbound can be calculated as:

```
   5     *    (    0     +    (     5          *          5    )    )

upperbound      number of       upperbound for      upperbound for
for the         facts in        the subgoal         the subgoal
subgoal         procedure       path(X,Y)           path(X,Y)
path            can_go        \--- the upperbound for the rule ---/
```

The result is 125. Usually, the actual number of iterations is much less than the upperbound. In this case, the number of iterations is only 41 with the solutions of $X = a, Y = b, Z = d;\ X = b, Y = c, Z = e;\ X = c, Y = d, Z = f$. In pure Prolog, if no recursive definition is present, each subgoal in a rule must be defined by a procedure with facts and/or rules which do not contain this subgoal. Since a fact is a rule with an empty body, it completely defines itself. Each subgoal in a rule must be eventually defined by some facts. Therefore, the upperbound for the number of iterations generated by each subgoal can always be calculated as above if the program is in pure Prolog and no recursive definition exists. Since there exists an upperbound for the number of iterations generated by backtracking in a non-recursive pure Prolog program, the search space for backtracking must have only a limited size.

On the other hand, if some terminating recursive definitions are present, there is still an upperbound for the number of iterations generated by backtracking. With the

introduction of recursive definitions, the iteration can no longer be calculated with the method described above. For example, in Program (a) in Figure 2.3, if the query of *?-can_go(X,Y)* is supplied, the method described above will calculate the upperbound as:

```
   0    +   (    5     *    (     0     +   (  5   * . . .
number of   upperbound for  number of facts  upperbound
facts in     the subgoal     in procedure    for the subgoal
procedure       path          can_go           path
can_go    \----------  the upperbound for the recursive rule  ----------/
```

It will result in an infinite sequence. We must determine the upperbound for the number of iterations generated by each recursive subgoal with another method instead of simply counting the number of the clauses in its corresponding procedure. Since the exact number of iterations that can be generated by a recursively defined subgoal is not known until semantic knowledge has been provided, there is no simple way to determine the exact upperbound for the number of iterations generated by these recursively defined subgoals. However, if the recursive definition involved can itself terminate

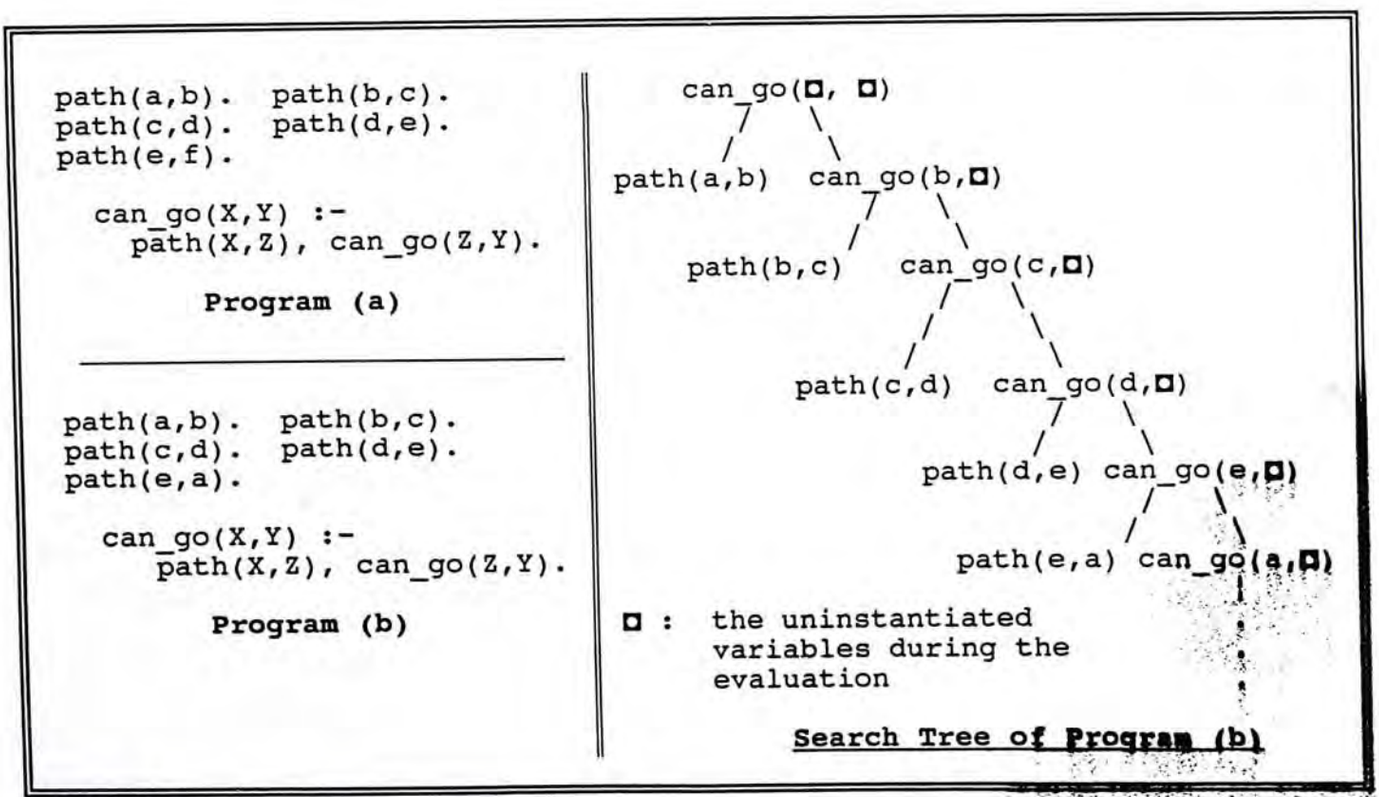


Figure 2.3

properly, we can arbitrarily assign the upperbound for the number of the iterations generated by these recursive subgoals to be any finite number, $\mathbf{n_i}$. For example, in Program (a), the recursive definition can terminate, so that the upperbound for the number of iterations that can be generated by the query *?- path(X,Y), can_go(Y,Z)* is the product of 5 and $\mathbf{n_1}$. It is still a finite number. Thus,the number of iterations generated by backtracking is still finite if the recursive definitions present can terminate properly.

But the iteration generated by backtracking can never stop if any nonterminating recursive definition is present. The number of iterations generated by a nonterminating recursive subgoal is infinite since no upperbound for the number of recursion levels implies that there is no upperbound for the number of iterations generated by a nonterminating recursive subgoal. The search space is infinite in this situation. Nontermination results. It can be shown by the search tree of Program (b) in Figure 2.3. Whatever the iteration is generated by the recursive definition or the backtracking process, the number of iterations can be infinite only if a nonterminating recursive definition is present. In conclusion, nontermination may occur only in a pure Prolog program with a recursive definition. However, some recursive programs can terminate while others cannot. Nontermination can be detected if we can find out the presence of a nonterminating recursive definition in the underlying program.

To have a recursive definition that terminates properly, two criteria must be satisfied at the same time:

(1) **There must exist some exit conditions.**

(2) **The recursive definition must be written in such a way that it can bring the execution to an exit condition at a certain point during the evaluation.**

As shown in the algorithm in Figure 2.1, the parameters in the input goal and the parameters in the head of those potential clauses play an important role in the unification process. Since the unification process can determine which clause to be chosen, it implies that parameters can greatly affect which clause to be selected at the next level of recursion. Moreover, recursion will stop when the unification of the recursive rule head and the recursive subgoal fails. In the following sections, we shall

discuss what an exit condition is in Prolog programs and how parameters can be related to the presence of nontermination errors.

2.3 Exit condition

There must be an exit condition to terminate the evaluation of a recursive definition. If after a finite number of recursion levels, the exit condition is reached such that the recursive path will not be visited again. Although how to define an exit condition in a recursive definition is similar among different conventional programming languages, how to define an exit condition in a Prolog program is greatly different. In conventional programming languages, the exit condition is always stated **explicitly**. However, as the Prolog execution model completely takes care of the control component of a program, an exit condition in a Prolog program can exist **implicitly** in two ways:

(1) **as one or more subgoals in the recursive rule that will fail at a certain point of the recursion,** or

(2) **as parameters in the parameter list of the recursive rule head or the recursive subgoal of a recursive rule that will cause the unification process to fail at a certain level of the recursion.**

In conventional programming languages, the recursive part and the non-recursive part of a recursive definition are always related to **mutually exclusive** conditions. In general, all recursive definitions that can terminate properly should have a structure similar to the one shown in Figure 2.4. It includes a condition and a decision control structure. In the case shown in Figure 2.4, once the exit condition is met, the procedure will be **prohibited** from recurring again and will terminate. Therefore, the condition specified **can serve as** an exit condition only with the co-existence of the *if-then-else* **control structure. The**

```
Procedure RRR (arguments)

 begin
  if exit condition
    then
      do non-recursive part
    else
      call RRR (modified arguments)
          {* recursive call *}
 end
```

Figure 2.4

control structure makes the recursive path and the non-recursive path mutually exclusive. Without such a control structure, the above condition cannot work as an exit condition.

Program (a)

```
sum (Var N, Result: integer);

Var M, R: integer;

Begin
  If N = 0
    {{ exit condition }}
    then Result := 0
    else begin
         M := N - 1;
         sum(M, R);
         Result := R + N
         end
End;
```

Program (b)

```
sum(N, 0) :- N == 0.

sum(N, Result) :-
     M is N - 1,
       sum(M, R),
         Result is R + N.
```

Figure 2.5

However, such **an Algol-like *if-then-else* control structure does not exist in pure Prolog**. The Prolog programmer has to consider how to achieve a mutually exclusive relationship between the recursive part and the non-recursive part by himself. As has been described above in Section 2.2, the pure Prolog programmer must let the Prolog determine the control. Nevertheless the unique backtracking mechanism in Prolog does not allow the clauses in the same procedure to be mutually exclusive. This can be illustrated by comparing the Pascal-like recursive procedure with its Prolog counterpart in Figure 2.5. In spite of their similar outlooks in logic, Program (a) in Figure 2.5 can terminate while Program (b) is a nonterminating Prolog program. The reason is that the two rules in the procedure *sum* of the Prolog program are not mutually exclusive under the backtracking mechanism. When Program (b) in Figure 2.5 is executed, the rule *sum(N,0) :- N == 0* will be reached after a finite number of recursions and will return a solution for the variable *Result* . However, it does not stop at this point, but

rather it goes to find further solutions due to the backtracking mechanism. This results in nontermination.

One simple way to achieve a mutually exclusive relationship between the recursive clauses and the non-recursive clauses is to add some extra subgoals in the recursive clause so that these subgoals will fail when the non-recursive clause is reached. For example, the second clause in Program (b) in Figure 2.5 can be re-written as:

sum(N, Result) :- N > 0, M is N - 1, sum(M, R), Result is R + N.

With the introduction of the subgoal $N > 0$ in the recursive clause, the recursive path and the non-recursive path now become mutually exclusive and termination can be ensured.

Since the absence of a mutually exclusive relationship between the recursive clauses and the non-recursive clauses can cause a recursive definition to be nonterminating, a diagnostic system that can identify this fault in a recursive definition can detect some kinds of nontermination errors in Prolog. It would be helpful to the Prolog novice who has programming experiences in conventional programming languages. In fact, it is quite easy to build a diagnostic system to detect the absence of such a mutually exclusive relationship if all the non-recursive clauses in the recursive definition are made up of facts only. First, the system should test whether the head of the recursive rule can unify with any of these facts. If any of them succeeds, the values specified in the fact are supplied to the recursive rule to instantiate the corresponding parameters in the rule. If none of the subgoals preceding the recursive subgoal fails in this process, the absence of a mutually exclusive relationship between the recursive and the non-recursive part is confirmed. For example, we can apply this technique to the program in Program (b) in Figure 2.5 with a minor adjustment. Without any change in the semantics of the program, the non-recursive clause is modified from *sum(N,0) :- N == 0* to a fact, *sum(0,0)* , to permit the application of this technique. Since the fact *sum(0,0)* can unify with the head *sum(N,Result)* , we proceed with the test and apply the values *0* and *0* to the parameters *N* and *Result* in the recursive rule. This results in:

sum(0, 0) :- M is 0 - 1, sum(M, 0), . . .

Because the subgoal preceding the recursive subgoal, *M is 0 - 1* does not fail, this shows that a mutually exclusive relationship is absent from this recursive definition. However, this simple technique becomes too weak when the non-recursive clauses are not all made up of facts. With this technique, we can conclude about the absence or presence of any mutually exclusive relationship only after we have known that all the possible values can be accepted by the non-recursive clauses and have tried these values on the recursive rule. If some rules exist in the non-recursive part as well, more time is needed to determine all these possible values.

On the other hand, there is another more fundamental problem in detecting the exit condition. Although it is a good practice for the Prolog programmer to ensure the presence of a mutually exclusive relationship between the recursive and non-recursive clauses, to achieve such a relationship **is not the same as supplying an exit condition to the Prolog program. With the backtracking mechanism in Prolog,** reaching a non-recursive clause **is not sufficient** to ensure the presence of an exit condition because it cannot prevent the recursion from taking place again in the recursive rules. When a non-recursive clause is reached after certain levels of recursion, the backtracking mechanism causes the recursion to resume at **the next clause** in **the procedure.**

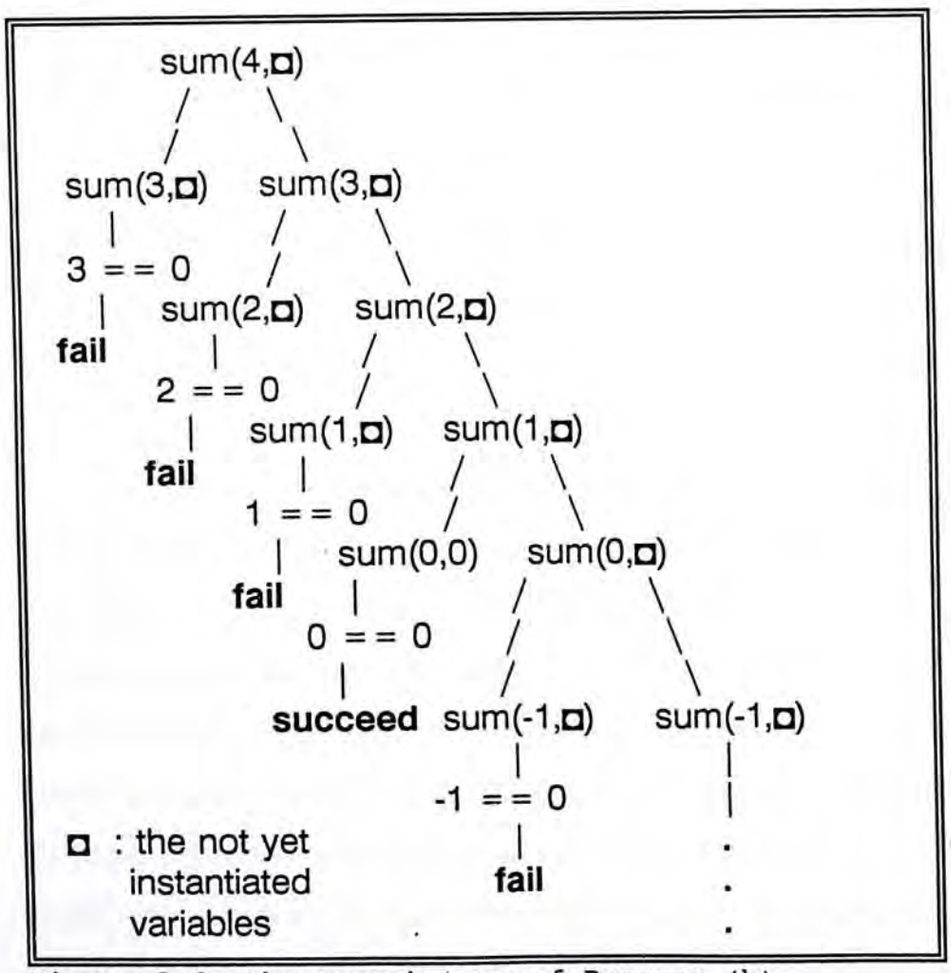


Figure 2.6 the search tree of Program (b) in Figure 2.5

Eventually, the recursive rule will be reached again. It can be illustrated by the search tree generated after the query of *?- sum(4,X)* is supplied to Program (b) in Figure 2.5. In Figure 2.6 we can clearly see how the Prolog execution model causes the recursive rule to be used again even when the non-recursive part has been reached. Moreover, reaching a non-recursive clause is **not a necessary condition** for termination in Prolog. It can be illustrated by the program in Figure 2.7. This program obviously does not have any non-recursive part for the recursive procedure *a_call* . However, it can terminate with any value supplied to the parameters X and Y . The reason behind is that the subgoal *not_exit(X,Z)* can stop the evaluation of this recursive definition

```
a_call(X, Y) :-  not_exit(X,Z),
                 a_call(Z,Y).

not_exit(1,2).
not_exit(2,3).
not_exit(3,4).
```

Figure 2.7 a recursive procedure without a non-recursive part

```
terminate([X|Y]) :-
        terminate(Y).
```

Figure 2.8 a terminating procedure using parameter as an exit condition

once its parameters cannot be instantiated with any one pair of the following values: *(1,2)* , *(2,3)* or *(3,4)* . The problem of this definition is not one of nontermination but rather missing solution.

Therefore, the non-recursive part of the definition cannot act as an exit condition by itself alone in Prolog, although it is important in finding a solution in the evaluation. Actually, an exit condition in a Prolog program is **any constraint that can stop the re-entrance of the recursive rule at a certain point of the evaluation of the recursive definition.** By examining the above examples carefully, we can discover that in order to have a recursive definition to terminate, a certain subgoal in the recursive rule needs to ultimately become unsolvable during the evaluation of this recursive definition. **Thus,** unlike the situation in conventional programming languages, the exit condition must be specified somewhere outside the recursive part, **the exit condition in Prolog programs must be present within the recursive clauses.** Usually, an exit conditions in a Prolog

recursive definition consists of one or more subgoals (actually, one subgoal is sufficient) which can become unsolvable at a certain point of the evaluation of the recursive definition. In some special situations, for example, when lists or structured data are used as the parameters in the recursive rules, as shown in Figure 2.8, they can cause unification of these recursive rules to fail and thus stop the next level of recursion. Therefore the parameters should also be considered alongside with the subgoal when trying to find an exit condition.

In the following examples, we further show that the exit condition and the mutually exclusive relationship between the recursive and the non-recursive part are

```
father(abraham,isaac).      ancestor(abraham,isaac).
father(isaac,  jacob).      ancestor(abraham,jacob).
father(jacob,  joseph).     ancestor(X, Z) :-
father(jacob,  judah).          father(X, Y),
                                    ancestor(Y, Z)
```

Figure 2.9 Example showing that termination can be achieved without the mutually exclusive relationship

independent to each other in pure Prolog. In Figure 2.9, no mutually exclusive relationship exists between the recursive clause and the non-recursive clauses. All the facts can unify with the recursive clause in the program. If the values *abraham, isaac* are used, the recursive clause will be instantiated as *ancestor(abraham, isaac) :- father(abraham, Y), ancestor(Y, isaac)* . Since the subgoal *father(abraham,Y)* in the instantiated recursive rule can succeed to unify with the fact of *father(abraham,isaac)* , the next level of recursion can continue with the recursive subgoal instantiated as *ancestor(isaac,isaac)* . On the other hand, if the values *abraham, jacob* are used, the recursive clause will become *ancestor(abraham, jacob) :- father(abraham,Y), ancestor(Y,jacob)* . Because of the fact *father(abraham, isaac)* , the subgoal *father(abraham, Y)* in the rule can again succeed. The recursive subgoal will become *ancestor(isaac, jacob)* and one more level of recursion can happen. Therefore, it is obvious that the recursive part and the non-recursive part of this recursive definition are not mutually exclusive. However, even though a mutually exclusive relationship does not

exist, the evaluation of this recursive definition can terminate. The evaluation of the recursive definition will eventually cause the subgoal *father* to be instantiated as either *father(joseph,Y)* or *father(judah,Y)* which cannot be unified with any clause in the program and thus this failure in unifying the subgoal with any *father* clause stops any further recursion. Again, it shows that it is the unsolvable subgoal in the recursive rule rather than the non-recursive part of the recursive definition that acts as an exit condition.

Furthermore, the existence of a mutually exclusive relationship cannot guarantee the existence of an exit condition because it is possible that the evaluation of such a

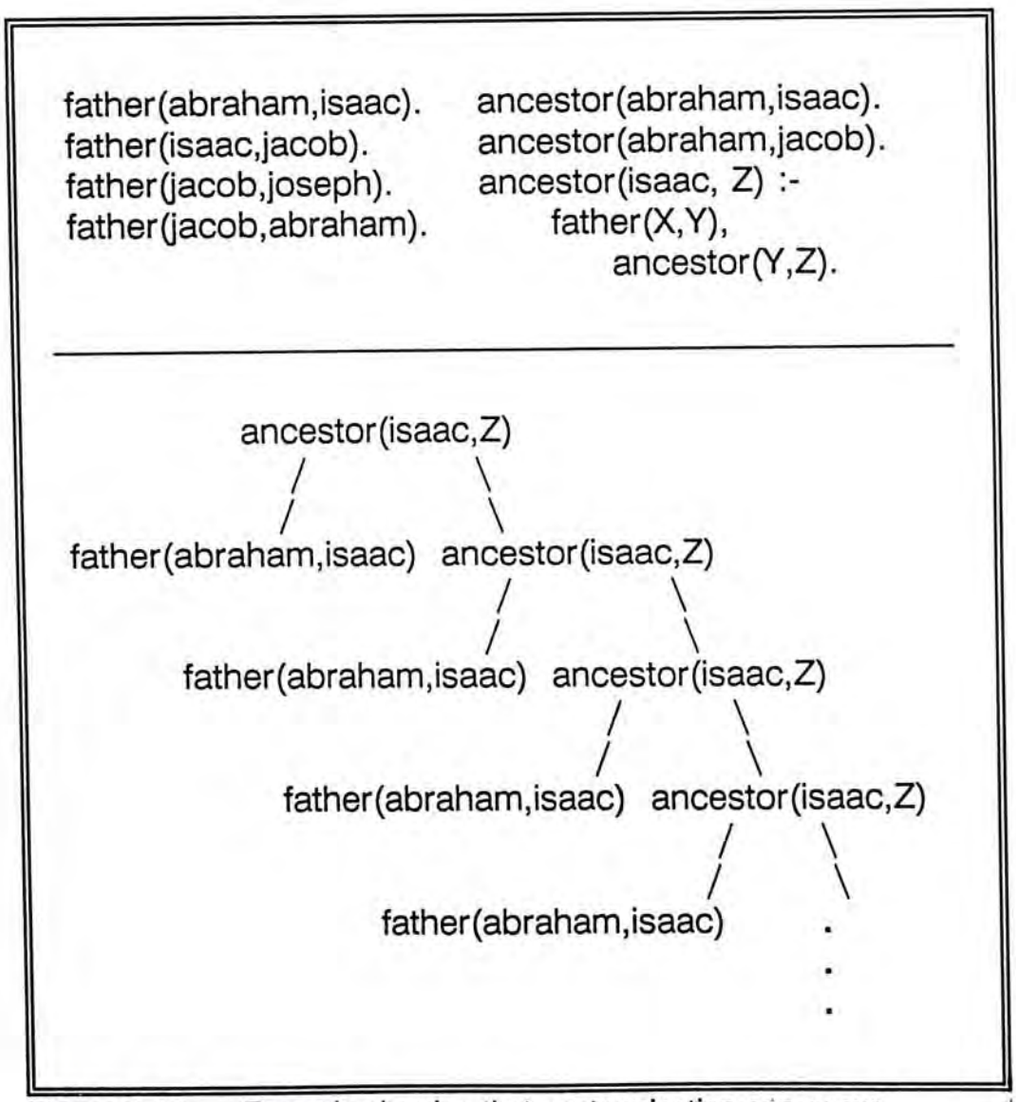


Figure 2.10 Example showing that nontermination can occur with the mutually exclusive relationship

recursive definition can never invoke those non-recursive clauses. The program in Figure 2.10 shows such possibilities. In this program, the facts *ancestor(abraham,isaac)* and *ancestor(abraham,jacob)* cannot unify with the head *ancestor(isaac,Z)* and thus they are mutually exclusive and thus there is no further evaluation of the recursive definition once one of the facts has been reached. However, the problem is that the facts can never be reached once the evaluation of the recursive definition starts to proceeded. If the query *?- ancestor(X,Y)* is supplied, after the two facts are reached, the backtracking mechanism causes the recursive rule also to be reached. As has been illustrated in the search tree in Figure 2.10, the recursive subgoal *ancestor* will always be instantiated as *ancestor(isaac, Z)* during the evaluation and thus the non-recursive clauses can never be reached. Nontermination then occurs. In fact, the mutually exclusive relationship between the facts and the recursive rule has no effect on the termination of a recursive procedure. Therefore, **the mutually exclusive relationship between the recursive part and the non-recursive part of a recursive procedure cannot guarantee proper termination.**

In conclusion, it is hard to determine which part in a recursive definition is an exit condition. Unlike the conventional programming languages, in which an exit condition is always stated **explicitly** as a part of the decision control construct, an exit condition in a Prolog program can only be established **implicitly** through some subgoals or parameters in the recursive rule only. On the other hand, a mutually exclusive relationship between the recursive clauses and the non-recursive clauses that is characteristic of an exit condition in conventional programming languages is neither sufficient nor necessary to establish an exit condition in Prolog. We therefore cannot use the mutually exclusive relationship as an indicator of an exit condition. Without any semantic knowledge, it is impossible to detect whether there is an exit condition. This difficulty has become a great barrier to the development of any analytical approach for nontermination detection.

However, even when the necessary semantic knowledge **is available, it cannot** definitely show whether the evaluation of a recursive definition **can terminate. It** becomes more obvious when we consider the situation in conventional programming

languages as an example. The explicitly stated exit condition in conventional programming languages cannot guarantee termination. Although a certain condition has been specified in a recursive definition as the exit condition, if the recursive definition is erroneously defined, the evaluation can never reach the intended exit condition. Therefore, nontermination can be avoided only if this recursive definition has both an exit condition and an exit-condition reaching process at the same time. We also need to investigate what constitute an exit-condition reaching process before we can provide a solution to all these problems. For briefness, **exit-reaching process** is used instead of exit-condition reaching process in the following descriptions.

2.4 Exit-reaching process

In addition to an exit condition, a process is required to bring the evaluation to reach the exit condition specified. To detect the presence of such an exit-reaching process, we must know how such a process is established in a recursive definition. By comparing Prolog with conventional programming languages, we discover that the exit-reaching process in Prolog programs is closely related to **those parameters in the recursive rule head** and **the recursive subgoal**. They are closely related due to two facts:

(1) **parameters are part of an exit condition**, and

(2) **parameters are used to pass values to the next level of recursion.**

In general, an exit condition usually contain a **variable** and an invariant. During the evaluation of a recursive definition, the value of the variable would be modified at the different levels of recursion while the invariant always remains the same. When the variable attains a particular relationship with the invariant, the exit condition becomes effective and thus stops the recursion. This can be illustrated by the Pascal-like program in Program (a) in Figure 2.5. This program has an explicitly stated exit condition $N = 0$, where N is the variable and 0 is the invariant. During the evaluation of this recursive definition, the variable is continuously modified by the statement $M := N - 1$. At the point where the specific relationship between the variable and the invariant,

i.e., $N = 0$, is achieved, the exit condition becomes effective and blocks the evaluation from going into the recursive part again. The recursion thus terminates. Similarly, termination can also occur if a specific relationship between variables is achieved. For example, the condition $N > X$ can serve as an exit condition in the below modified version of Program (a) in Figure 2.5.

```
sum(Var N, X, Result: integer):

Var M, Y, R: integer;

Begin
If N > X
  Then Result := 0
  Else Begin
       M := N - 1;
       Y := X + 1;
       sum(M, Y, R);
       Result := R + N
       End
End;
```

Since an exit condition must contain such a variable to be effective, this variable can be referred to as an **exit-variable**. Hence, an exit-reaching process can be established only if two conditions are both fulfilled:

(1) **the recursive definition must be defined in such a way that the exit-variable can be modified during the evaluation**; and

(2) **the exit-variable must be modified in a direction in which the specific relationship between the variable and the invariant or between variables can be achieved.**

If condition (1) cannot be met, the evaluation of this recursive definition will certainly result in nontermination. The exit-variable remains the same during the successive levels of recursion and thus the exit condition can never be met. However, condition (1) only guarantees the existence of an **exit-variable modifying process. If the exit-variable**

modifying process cannot modify the exit-variable to reach the exit condition, the evaluation will end up in nontermination as well.

Obviously, the fulfilment of condition (2) requires the fulfilment of condition (1) as a prerequisite. If condition (1) cannot be satisfied in a recursive definition, condition (2) will also be absent in the definition. Thus the detection of an exit-reaching process can be conducted in two steps. Firstly, we detect whether an exit-variable modifying process exists. If it does not exist, we can conclude that condition (2) does not exist either and the exit-reaching process is absent. Hence nontermination is detected. However, when condition (1) is found, we need to further examine whether the exit-variable modifying process can act as an exit-reaching process. In other words, **the exit-variable modifying process is a potential exit-reaching process**. To detect the presence of an exit-reaching process, we first need to have a method to find the potential one. Then we also need another method that can verify whether the potential process found is an actual exit-reaching process. So we can detect an exit-reaching process only if we have methods to detect the two cases.

In conventional programming languages, it is quite easy to identify the exit-variable and the invariant used because the exit condition is stated explicitly. Since condition (1) demands an exit-variable modifying process in the recursive definition, the exit-variable must appear somewhere before the point at which the recursive call is invoked. If such a variable is absent, it indicates that condition (1) cannot be met and nontermination will surely occur. In Figure 2.11, we use a Pascal-like program again to give a clearer illustration. The exit-variable and the invariant are also N and 0 respectively. Although it is similar to Program (a) in Figure 2.5, it cannot terminate. By examining this program carefully, we can see that the variable N does

```
sum (Var N, Result: integer);

Var M, R: integer;

Begin
  If N = 0
    {{ exit condition }}
    then Result := 0
    else begin
         M := M - 1;
         sum(M, R);
         Result := R + N
         end
End;
```

Figure 2.11

not appear before the recursive call. Therefore, the variable can never be modified during the recursion. Nontermination happens due to the absence of an exit-variable modifying process.

It follows that the method to detect an exit-variable modifying process is quite simple. First, we identify all the possible condition variables. By analyzing the recursive definition, we can know whether these variables appear in any statements before the recursive call. If they are found in some statements, we then examine the recursive call to find what variables are used and whether these variables also appear in the same statement. In Program (a) in Figure 2.5, we first examine whether the variable N exists before the recursive call. Then we examine what variable is used in the recursive call, which is M in this case. Since both of them exist in the statement $M := N - 1$, an exit-variable modifying process is considered to be found. We shall find that the same method can be applied to Prolog programs.

In Prolog, as discussed in Section 2.3, the exit condition is stated implicitly through certain subgoals in the recursive rule or some parameters in the head of the recursive rule. To simplify the discussion, we first consider **the case of the exit condition that is formed by parameters only**. If the example in Figure 2.8 is examined, we can see that nontermination occurs if an uninstantiated variable is supplied to the parameter *[X|Y]*. This situation in fact is part of the well-known *occur check* problem. However, if any value or instantiated variable is supplied, the parameters in the head of the recursive rule in Figure 2.8 can act as an exit condition because the parameters can be unified with only a certain range of values; in this case, it is any non-empty list. Once the value supplied is not in this range, i.e., the empty list in this example, the unification of this recursive rule fails and the next level of recursion is denied. Hence, for the program in Figure 2.8, we can consider that the exit-variable is the parameter ***[X|Y]*** while the invariant is the empty list *[]*. Obviously, the particular relationship between the exit-variable and the invariant that needs to be achieved to exit the recursion is:

the value supplied for the parameter is an empty list.

```
a(1,0).

a(f(X), N) :- a(X, N)
```

Figure 2.12

Although the parameter does not appear in the body of the recursive rule, the *head-tail separator* in the parameter *[X|Y]* does the job of the exit-variable modifying process so that condition (1) is met. Therefore, the parameter *[X|Y]* does not only constitute an exit condition but also an exit-variable modifying process. Furthermore, the *head-tail separator* always cuts one element from the head of the list at each level of recursion. Eventually all the elements will be taken away. An empty list will result and thus the exit condition is met. It satisfies condition (2). Thus, it also works as an exit-reaching process. Besides the list, structured data can also be used as a parameter in the head of a recursive rule to have similar effect. Similarly, we can consider the parameter *f(X)* in Figure 2.12 to be the exit condition. When a value supplied to the parameter is not a structured data with the predicate name *f*, the next level of recursion will be blocked. On the other hand, the parameter itself can be considered as the exit-variable while the invariant is any term which is neither a variable nor a compound term with predicate name *f*.

```
a(1,X,0).
a(0,X,N) :-
   modify(X,Y,Z), a(Z,Y,N).

modify(3,2,0).
modify(2,1,0).
modify(1,3,1).
```

Figure 2.13

However, if other data structure is used as the parameters in the head of the recursive rule, the result is quite different from the above two cases. When a certain constant is used, as what is illustrated in Figure 2.13, it can still work as an exit condition in some situations. In Figure 2.13, the first parameter in the recursive rule head is *0*, thus the unification fails when any value other than *0* is supplied. Therefore the value *0* is the invariant of the exit condition. But what is the exit-variable? By analyzing the recursive definition, one will see that the first parameter cannot act as an exit-variable because its value is already fixed to be *0*. Actually, the exit-variable is in the second parameter. The exit-variable modifying process is provided by the subgoal *modify(X,Y,Z)*. Since the first parameter of the recursive subgoal is *Z* and the only parameter from the head is

X in the subgoal *modify* , we can conclude that the second parameter X in the head acts as the exit-variable.

Program (a)

```
a(1,0).
a(X,N) :-
   modify(X,Y), a(Y,N).

modify(1,2).
modify(2,3).
modify(3,4).
```

Program (b)

```
a(1,0).
a(X,N) :-
   modify(X,Z), a(Y,N).

modify(1,2).
modify(2,3).
modify(3,4).
```

Figure 2.14

Through the discussions of the different cases of using parameters as the exit condition, we can see how the exit-variable modifying process, or even the exit-reaching process, is closely related to the parameters used in the recursive rule head and the recursive subgoal. Although it demands different skills to state an exit-variable modifying process with different types of parameters, we can see that the presence of an exit-variable modifying process can be detected by analyzing the parameters involved. It is due to the fact that the parameter is a part of the exit condition. In Prolog, the exit-variable, and sometimes the invariant, is constituted by the parameter.

In **the case of the exit condition that is made up of subgoals**, parameters play a significant role as well. In Program (a) in Figure 2.14, since all the parameters in the head of the recursive rule are variable, they can no longer act as an exit condition. This recursive definition can terminate because of the subgoal *modify(X, Y)* . Since the first parameter of the subgoal *modify* can only accept the values *1, 2, 3* , any value out of this range can cause it to fail and stop further levels of recursion. Moreover, the subgoal *modify* plays the role of an exit-variable modifying process. It is responsible for the change of the first parameter in each level of recursion. By comparing it to Program (b) in Figure 2.14, which does not terminate, we can see the important role of a parameter in the exit-reaching process. By changing the subgoal *modify(X,Y)* to *modify(X,Z)* , the exit-variable modifying process is destroyed. In this case, its importance is due to the fact that these parameters are responsible for passing data from one level of recursion to the next level. If the linkage

is broken, the exit condition can never be reached. Therefore, an exit-variable modifying process in a Prolog program can be identified with a similar method applied in conventional programming languages. First we analyze the subgoal preceding the recursive subgoal. If there exist some subgoals that have the parameter corresponding to the exit-variable, we can consider the exit-variable modifying process to be present. In Program (b) in Figure 2.14, since the parameter Y in the recursive subgoal is not present in the subgoal *modify* , an exit-variable modifying process cannot be established.

Hence, it is possible to have a method to detect an exit-variable modifying process through the analysis of parameters involved in the recursive rule. On the other hand, it is difficult to verify whether an exit-variable modifying process is also an exit-reaching process. Semantic knowledge is needed to determine the direction of the evaluation of the recursive definition that will be brought about by the exit-variable modifying process found. However, in pure Prolog, because no built-in predicate exists, every term should be defined in the program. It allows us to develop a method to examine what kind of data would be passed through the exit-variable modifying process during the successive levels of recursion. In Chapter 4, we shall show how one can verify an exit-variable modifying process through data analysis.

2.5 Parameter Based Detection

In considering how to terminate a Prolog recursive definition, two aspects of the termination requirement can be identified:

(1) exit conditions and

(2) an exit-reaching process.

However, **both of them must co-exist in one recursive definition** in order to make this definition terminate properly. When considering what an exit condition is, we can see that the presence of an exit condition implies the presence of an exit-reaching process. The absence of a mechanism in a recursive definition to direct its evaluation to the

supposed exit condition is equivalent to the absence of an exit condition in this recursive definition. On the other hand, the presence of an exit-reaching process implies that there exists at least one exit condition for the process to reach. Therefore, they are in an **interdependent** relationship.

Because of this interdependent relationship, any nontermination detection method which considers only one aspect of the termination requirement or tries to handle them separately cannot succeed. Moreover, because an exit condition is stated implicitly in Prolog, it seems to require semantic knowledge for detecting exit conditions. It becomes a great problem in the attempt to develop an approach to detect nontermination in Prolog.

However, the interdependent relationship also implies that a method that can detect an exit condition can also detect an exit-reaching process or vice-versa. In Section 2.4, we show the possibilities to detect an exit-variable modifying process, i.e., the potential exit-reaching process, through the analysis of the parameters in the recursive rule. Because of the interdependent relationship between an exit condition and an exit-reaching process, the task of detecting nontermination in Prolog programs can be accomplished **if we can have a method to identify an exit-variable modifying process and then a method to verify whether it is also an exit-reaching process.**

Therefore, we propose a nontermination detection technique based on parameter analysis as follows:

(1) By analyzing the parameters, we first find the potential exit-variable modifying process. In Prolog, since all the exit-variables are made up of the parameters **in** a recursive subgoal, to detect a potential exit-variable modifying process **is** equivalent to detect **a parameter modifying process**

(2) If a parameter modifying process exists in a recursive definition, it can **be shown in pure Prolog** that there also exists a technique that can detect whether **a Prolog** program will terminate **without any prior knowledge of the presence of an exit-condition.**

Since the absence of any parameter modifying process implies the absence of an exit-variable modifying process, this can show how the detection of a parameter modifying process plays a role in nontermination detection in Prolog. Actually, as we shall see later, the detection of a parameter modifying process constitutes the basis of our non-tracing nontermination detection technique. Therefore, our technique for nontermination detection consists of two parts: detection of parameter modifying processes and verification of the detected parameter modifying processes. The first part of our technique will be discussed in Chapter 3. We shall explore how to detect the parameter modifying process in Prolog programs. The second part of the technique is based on an analysis of data passing through the parameters involved in the parameter modifying process. In a pure Prolog program, it can be shown that it is possible to know what values can pass through the parameters in a parameter modifying process merely by analysis. It will be discussed in Chapter 4 in detail. However, a **program structure analysis** must first be performed to identify all the recursive definitions in a Prolog program before parameter analysis and data analysis can be conducted. As implied by the above discussion, an exit-reaching process only exists in a recursive definition. Therefore, only the recursive definitions require our concerns in nontermination detection and parameter analysis and data analysis should be conducted on them alone. Since it is trivial to detect a recursive definition, no detail on recurisve definition detection algorithms is given in the thesis.

In conclusion, our analysis of the relationship between recursive definitions and nontermination is intended to develop a new nontermination detection technique in pure Prolog. With the emphasis of the relationship between the exit-reaching process and nontermination, it can be shown later that this provides a new starting point to detect nontermination in pure Prolog programs. In the following chapters, we shall see how an exit-reaching process can be detected step by step by analyzing the parameters and data used in the recursive definitions of a Prolog program. Therefore, our discussion above provides a basis to develop a technique based on a compile-time program structure analysis approach instead of the run-time tracing technique. Moreover, understanding of the cause of nontermination in pure Prolog programs gives us insights in developing methods to overcome the limitation of the run-time tracing technique.

CHAPTER 3 —Parameter Analysis

To detect whether a parameter modifying process is present in a certain recursive definition, an appropriate technique is needed. However, a parameter modifying process cannot be detected by analyzing the parameters alone. To detect whether values in a parameter are modified during recursion, one needs to analyze the values transferring in the parameter too. In this chapter, however, a technique, **parameter analysis**, which can identify a set of **potential parameter modifying processes** by analyzing the parameters alone is developed. It is based on detecting some unique characteristics of potential parameter modifying processes that are reflected in the relation among the different parameters in a recursive definition. The analysis consists of two steps:

(1) to detect the presence of any **parameter links** in a certain recursive definition, and

(2) to verify whether any parameter link found is a **cyclic parameter link**.

In Section 3.1, we shall discuss what a parameter link is in a pure Prolog program and how it is related to the parameter modifying process. In Section 3.2, we shall further see how a parameter link can become a cyclic parameter link and why a cyclic parameter link is a potential parameter modifying process, which must exist in a recursive definition in order to have the presence of a parameter modifying process. After discussing what constitutes the characteristics of a potential parameter modifying process, we introduce our systematic approach to detect the process through graphical representation of parameter links. A set of graphic notations is introduced to explain how the technique works. Finally algorithms are also provided to show how one can detect a potential parameter modifying process in general.

3.1 Parameter Link

The concept of parameter link is developed as a preliminary step for identifying a potential parameter modifying process. **A parameter link is considered to be present between a parameter in the head of a certain recursive rule and a parameter in the corresponding recursive subgoal if the value of the parameter in the recursive subgoal is dependent on the value of the parameter in the recursive rule head.** When there is a dependent relationship between a parameter in the recursive rule head and a parameter in the recursive subgoal, the value used in the parameter in the recursive rule head will affect the value assigned to the parameter in the recursive subgoal. Due to the characteristics of the Prolog execution model, the same variable in a rule cannot be instantiated to different values. If we want to receive a value from a previous level of recursion and then modify it and pass it to the next level, we must have at least two different variables that have a dependent relationship. Therefore, this dependent relationship between two parameters can also be described by a **data transfer analogy**. This analogy will be used to facilitate our discussion. In the following sections, we shall discuss how a parameter link relates to a parameter modifying process and how a parameter link can be identified in different kinds of Prolog recursive definitions.

3.1.1 Parameter Link and Parameter Modifying Process

As discussed in Section 2.4, an exit-variable needs to be a parameter in Prolog programs. In other words, a parameter modifying process is a potential exit-variable modifying process. Since an exit-reaching process implies the presence of an exit-variable modifying process, a method detecting parameter modifying process can provide potential exit-reaching processes for our further analysis. The concept of parameter link is thus introduced to facilitate the detection of parameter modifying process. To see how a parameter link relates to a parameter modifying process, we must first understand fully what constitutes a parameter modifying process. A parameter modifying process consists of two aspects:

(1) it can modify a parameter in a recursive definition at each level of recursion, and

(2) the effect of the modification can be passed along to successive levels of recursion **continuously**.

Aspect (1) of the parameter modifying process is obvious. As a parameter modifying process, it must be capable modifying a parameter at every level of recursion. However, aspect (2) needs some further elaboration. It will become clearer if we consider the opposite case.

In a recursive definition, there may exist a process that can modify a parameter at each level of recursion separately. But this process does not really modify the involved parameter if the recursive definition is considered as a whole. This can be shown by the example in Figure 3.1. In Figure 3.1, the subgoal *modify(Variable, New_Variable)* is intended to modify the parameter *variable*. Because the variables with the same name in the same rule cannot be instantiated to different values in Prolog, another variable *New_Variable*, instead of the original *Variable*, is used to contain the result of the modification.

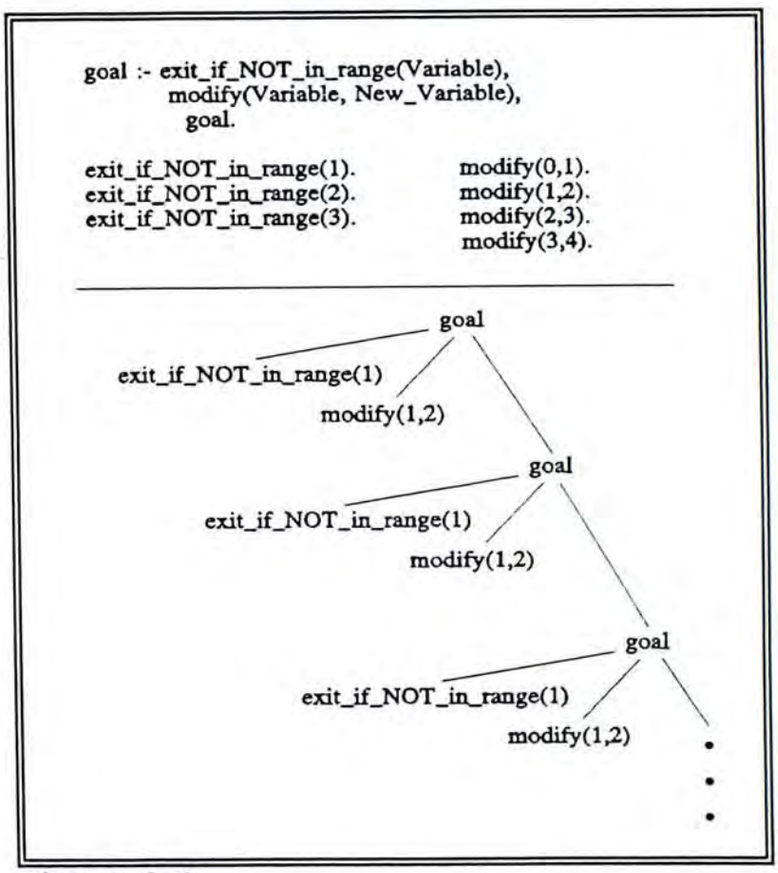


Figure 3.1

Although the subgoal *modify(Variable, New_Variable)* can modify the parameter *Variable* at each level of recursion, *Variable* still remains the same throughout the evaluation as shown by the search tree in Figure 3.1. The reason is obvious: since the parameter *New_Variable* which contains the result of the modification does not affect any parameter in the recursive subgoal, the change made on the parameter *Variable*

cannot be passed onto the next recursion level. As a result, all the parameters in the recursive definition are instantiated to same values during every level of recursion. Thus nontermination results. This is the reason of why a parameter modifying process cannot really fulfill aspect (1) without fulfilling aspect (2). No parameter can really be modified if aspect (2) cannot be met.

However, aspect (2) of the parameter modifying process requires that there exists at least a process which can pass data from the previous level of recursion to the parameter modifying process and subsequently can also pass the result of the parameter modifying process to the next level of recursion. Such a process can be achieved if:

(1) a parameter can be supplied with a value from the previous level of recursion, and

(2) the value of the modified parameter can be passed onto the next level of recursion.

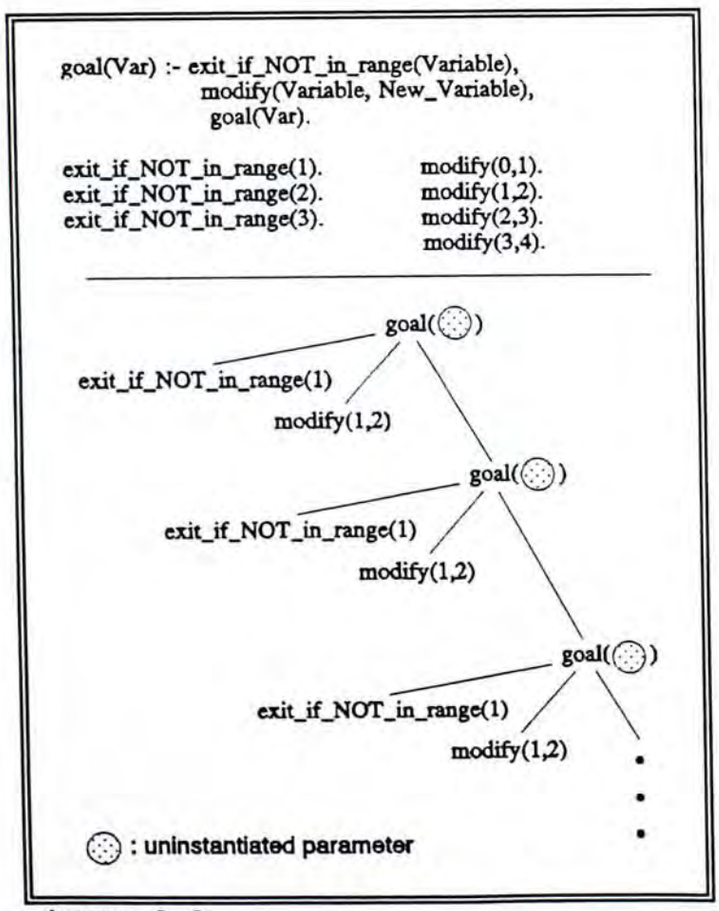


Figure 3.2

To have a value supplied from the previous level, some parameters must appear in the head of the recursive rule. Similarly, to have a value passed onto the next level of recursion, there must be requires some parameters in the recursive subgoal. The result of the absence of any parameter in the recursive subgoal has also been shown in Figure 3.1. But the presence of parameters in the recursive rule head and the recursive subgoal cannot guarantee the presence of a parameter modifying process in the recursive definition. In Figure 3.2, there is a recursive definition similar to the one in Figure 3.1 except that there are some parameters in the recursive rule head and the recursive subgoal.

Although there exists a parameter *Var* in the recursive rule head and the recursive subgoal, *Var* is not related to any parameter appearing in the recursive rule body. Therefore, the result of the modification can never be passed onto the next level of recursion. Moreover, the result of the modification from the previous level cannot affect the modification of any parameter in the present level of recursion. This is shown by the search tree in Figure 3.2. Parameters are instantiated to the same values if the parameters in the recursive rule head and the recursive subgoal do not related to the parameters in the recursive rule body.

Since a parameter link indicates the presence of a dependent relationship between a parameter in the recursive rule head and a parameter in the recursive subgoal, a parameter link can indicate that values can be received from the previous level of recursion and subsequently some modified values can be passed onto the next level of recursion from one level of recursion. Although the presence of a parameter link itself cannot completely satisfy the requirements of aspect (2) of the parameter modifying process, the existence of a parameter link in a recursive definition implies that *a parameter modifying process of one level of recursion* exists in the recursive definition. Therefore, **a parameter link is a necessary condition for the presence of the parameter modifying process**. That is because the effect of modification can be passed onto successive levels of recursion continuously only if there at least exist some mechanisms to receive and to pass on the values in each level of recursion. Without a parameter link, no parameter modifying process can be present. In other words, parameter links serve as an indication of all the possible parameter modify processes in a recursive definition.

With respect to aspect (1) of the parameter modifying process, parameter links also cannot completely satisfy its requirement. Detecting a parameter link, we can only conclude that the two parameters from the recursive rule head and the recursive subgoal are in a dependent relationship. But a parameter link does not guarantee that the content of involved parameters must be changed during recursion. For example, if the procedure defining the subgoal *modify(Variable, New_Variable)* is changed to a procedure consisting of only one fact: *modify(1,1)* in the above two examples, the values

instantiated to the parameters *Variable* and *New_Variable* are all remained as *1* even though a parameter link can be established in the recursive definition. We can be sure that parameters involved in a parameter link can be modified during the recursion only after we have analyzed the data transferring through the parameter link. However, parameter links in a recursive definition provide us with a pool of parameters that involve in the potential parameter modifying process. Thus parameter links can serve as an indicator of parameter modifying processes. Moreover, since a parameter modifying process is a potential exit-variable modifying process, the parameters involved in parameter links are also the potential exit-variables in a recursive definition.

3.1.2 Parameter Links of Multi-Parameters

Sometimes, a parameter link is not obvious. A parameter link can also be established between a parameter in the recursive rule head and a parameter in the recursive subgoal through a number of other parameters in the rule. In Figure 3.3, both the parameter *Variable*, which is modified by the subgoal *modify*, and the parameter *New_Variable*, which contains the result of the modification, do not appear as any parameter in the recursive rule head or the recursive subgoal. Yet this recursive definition includes a parameter modifying process. We can see this clearly if we consider it in terms of the dependent relationship. In the recursive definition in Figure 3.3, the parameter in the recursive subgoal, *A*, is dependent on the parameter *New_Variable* because of the subgoal *link2(New_ Variable,A)* while *New_ Variable* is also determined by *Variable*. Moreover, the parameter *Variable* depends on the parameter *X* through the subgoal *link1(X, Variable)* where *X* is the parameter in the recursive rule head. Thus, *A*, the parameter in the recursive subgoal, can be considered to be dependent on *X*, the parameter in the recursive rule head. A parameter link exists in this recursive definition between parameters *A* and *X* though it is established through a transitive relationship.

```
goal(4, exit).
goal(X,Y) :- link1(X, Variable),
             exit_if_NOT_in_range(Variable),
             modify(Variable, New_Variable),
             link2(New_Variable,A),
             goal(A, Y).

exit_if_NOT_in_range(1).        modify(0,1).
exit_if_NOT_in_range(2).        modify(1,2).
exit_if_NOT_in_range(3).        modify(2,3).
                                modify(3,4).

link1(X,X).                     link2(Y,Y).
```

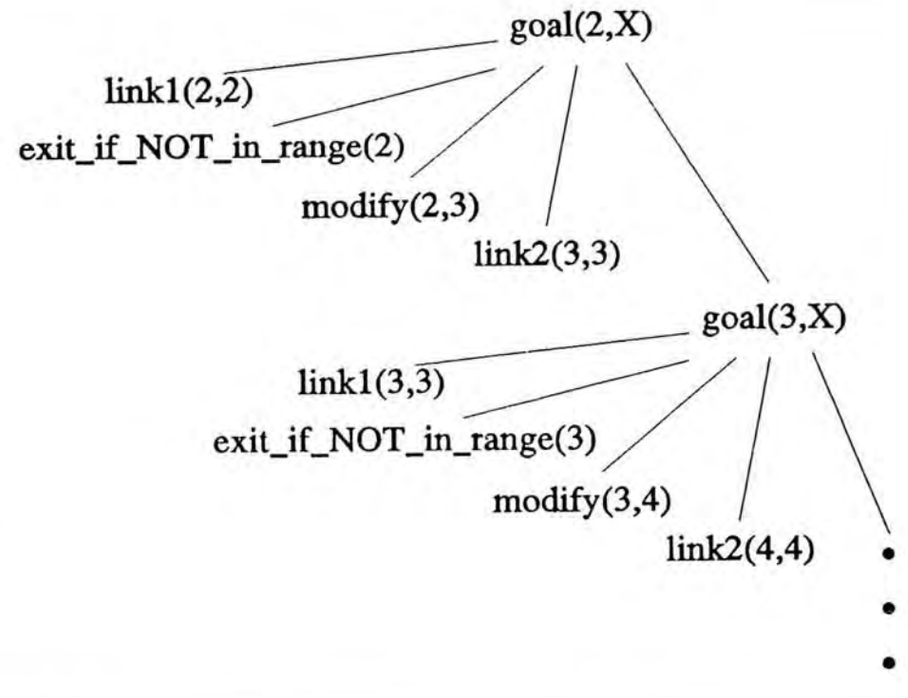


Figure 3.3

3.1.3 Parameter Links in Indirect Recursive Definition

So far, we have focused our discussion on direct recursive definitions only. We must also consider the case of indirect recursive definitions to have a complete picture of parameter links. In Figure 3.4, there is an indirect recursive definition. To solve *goal1*, the subgoal *goal2* must be evaluated. But the subgoal *goal3* must be first evaluated to solve *goal2*. Then the evaluation of *goal3* requires *goal1* to be solved. Illustrating this by the search tree in 3.4, we can see clearly how these three rules indirectly define a recursion. Moreover, the search tree in Figure 3.4 also shows that a parameter modifying process is present in this indirect recursive definition throughout the

successive levels of recursions. If the recursive definition is examined, we can see that, in the recursive rule *goal1*, there is a dependent relationship between *X*, a parameter in the recursive rule head, and *A*, a parameter in the recursive subgoal. Rules *goal2* and *goal3* are in a similar situation. By applying the definition of parameter link provided above to these cases, we can conclude that a parameter link exists in the rules of this indirect recursive definition. Furthermore, because the parameter link in rule *goal1* passes data to rule *goal2* through the first parameter in the recursive subgoal while the parameter link in rule *goal2* receives data from rule *goal1* through the first parameter in the head, data can be passed from the parameter link in rule *goal1* to the parameter link in rule *goal2*. We can consider that the parameter link in rule *goal1* is linked to the parameter link in rule *goal2*. Again, the same situation happens between the rules *goal2* and *goal3* and also the rules *goal3* and *goal1*. Since there exists a dependent relationship between a parameter in the head of rule *goal1* and a parameter in the recursive subgoal of rule *goal3*, a parameter link is present in this indirect recursive definition. It is similar to the case of direct recursive definition. In summary, a parameter link exists in an indirect recursive definition if:

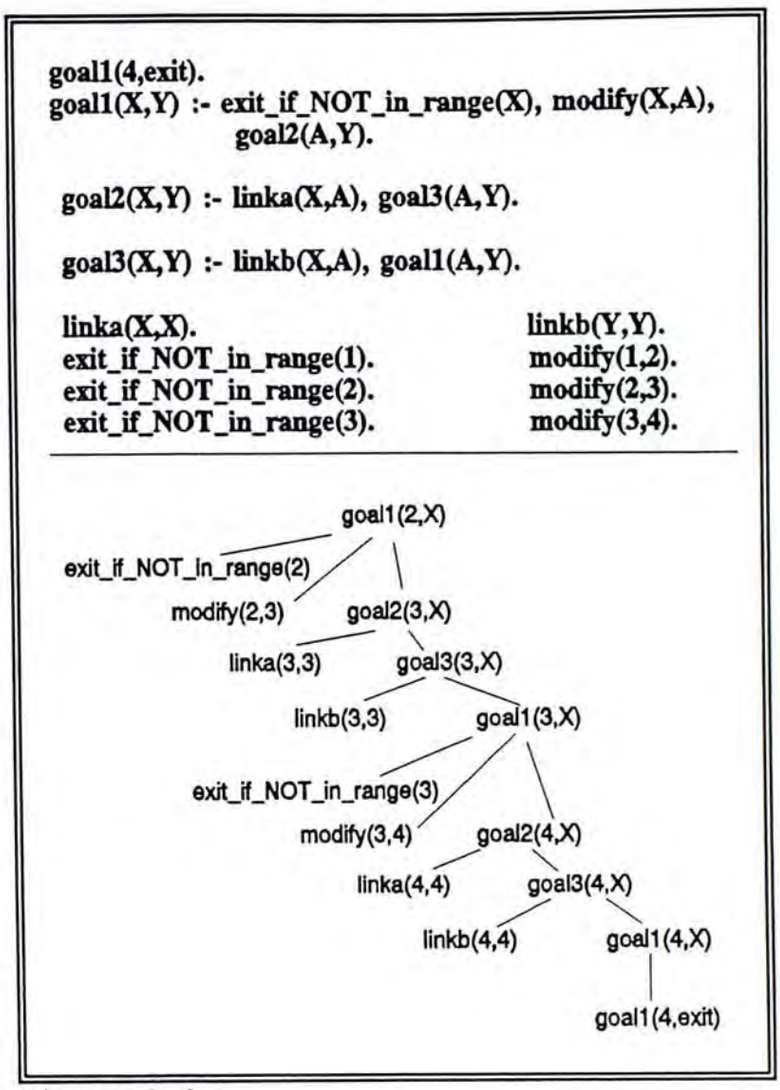


Figure 3.4

(1) parameter links exist in each rule involved in the indirect recursive definition, and

(2) at least one of the parameter links in each rule can be linked to the parameter link in the rule corresponding to the recursive subgoal.

3.1.4 Parameter Links with Special Parameters

Moreover, in pure Prolog, a parameter link can be actually formed in three ways:

(a) through some subgoals,
(b) through some special parameters, or
(c) through some subgoals and some special parameters.

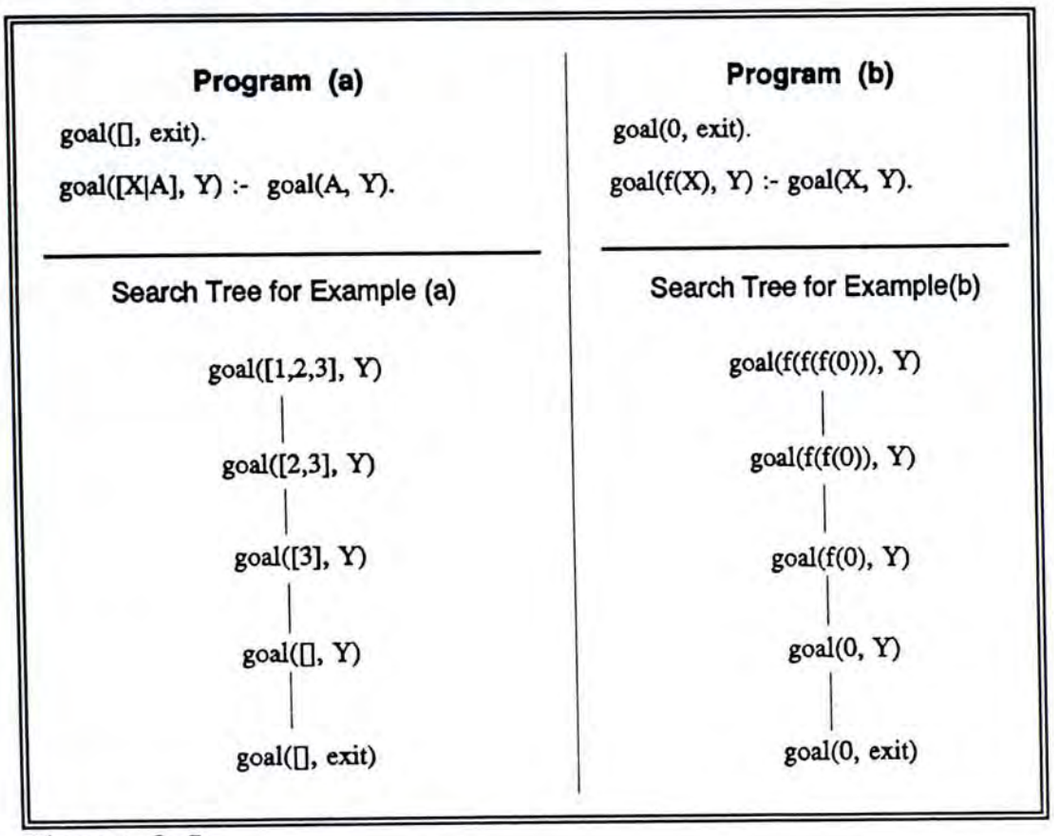


Figure 3.5

In the above examples, all the parameter links found are formed by subgoals. We shall refer to them as **subgoal parameter links**. In Figure 3.3, the parameter link is formed by a sequence of subgoals: *link1(X, Variable)*, *exit_if_NOT_in_range(Variable)*, *modify(Variable, New_Variable)* and *link2(Variable, A)*. In Figure 3.4, the sequence of subgoals that form the parameter link are *exit_if_NOT_in_range(X)* and *modify(X, A)* in rule *goal1*, *linka(X, A)* in rule *goal2*, and *linkb(X, A)* in rule *goal3*. But a parameter link can be established through some special parameters using lists and structured data. In Program (a) in Figure 3.5, we can see that, due to the *head-tail separator*, only the tail part of the list in the first parameter in the rule head can be passed onto the next level of recursion through the first parameter in the recursive subgoal. Therefore, it can be considered as a parameter link between the first

```
goal([], odd).
goal([X], even).
goal([X|A], Y) :- exit_if_NOT_in_range(A), modify(A,A_), goal(A_, Y).

exit_if_NOT_in_range([]).
exit_if_NOT_in_range([X|Y]) :- at_least_one_element(Y).

at_least_one_element([X]).
at_least_one_element([X|L]).

modify([X|L], L).
```

Figure 3.6

parameter in the recursive rule head and the first parameter in the recursive subgoal. A similar situation happens in Program (b). In this case, structured data are used to achieve a parameter link. However, subgoals and special parameters can also be **mixed** together to build a parameter link. In Figure 3.6, due to the list used in the first parameter in the recursive rule head, the first parameter in the head is linked to the parameter A in the rule body. Then A is linked to $A\_$ through the subgoal *modify(A,A_)* where $A\_$ appears as one of the parameters in the recursive subgoal. Therefore, a parameter link is established using both subgoal and special parameter. In these cases, we shall refer to the parameter links as **special parameter links**.

3.1.5 Parameter Links of the Same Name Parameters

In addition to subgoals and special parameters, a special case should also be considered when we construct parameter links through different data structures. It is the case of **parameters with the same name in the same rule**. We can consider that a parameter link exists in both Program (a) and Program (b) in Figure 3.7 because data can be passed from the recursive rule head to the recursive subgoal in both cases. By the search tree of Program (a) in Figure 3.7, we can clearly see that a value being passed onto the recursive definition by the parameter X can surely be passed through the successive levels of recursion. It is due to the fact that the first parameter in the recursive rule head and the first parameter of the recursive subgoal share the same

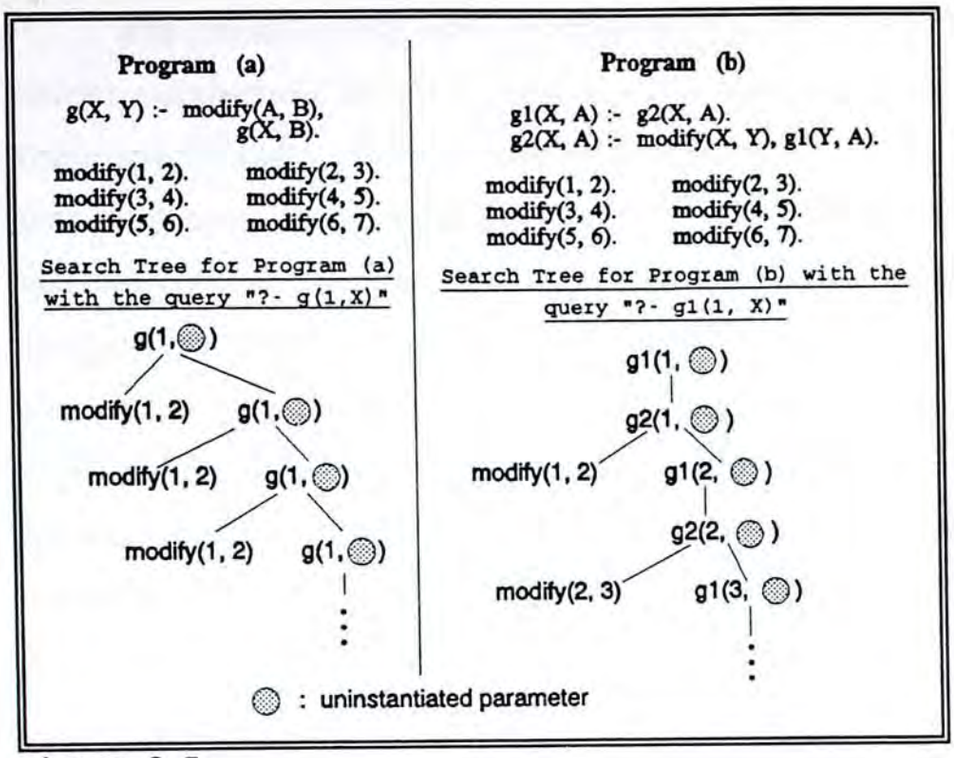


Figure 3.7

name. This is also true for the rule *g1* in Program (b).

However, a parameter link formed by common name parameters can never make any modification on the data passing these parameters since the parameters of the same name in the same rule must be instantiated to the same value in Prolog. Since modification of the transferring values (which is required by aspect (1) of a parameter modifying process) is absent in this kind of parameter links, the two same-name parameters in the recursive rule head and in the recursive subgoal cannot form a parameter link. So Program (a) is considered to have no parameter link. But Program (b) illustrates a different situation, where a parameter link is considered to be present.

The two parameters sharing the same name in the rule *g1* are important to establish the parameter link in Program (b). In the rule *g2* of Program (b), the first parameter in the rule head forms a parameter link to the first parameter in the recursive subgoal through the subgoal *modify*. But a parameter link cannot be established in this indirect recursive definition if there is no parameter link in the rule *g1*. In this case, the parameters sharing the same name in the rule *g1* provide a channel for passing on the modified data from the rule *g2* of the last level of recursion to the rule *g2* of the next level of recursion. In this situation, the same-name parameters in the rule *g1* must be considered to form a parameter link in the rule *g1* so that we do not miss the parameter link existing in the whole indirect recursive definition.

The situation illustrated in Program (b) is only one of the many special cases in which a parameter link can be considered to exist between two same-name parameters. These special cases can only arise in a recursive definition with other parameter links formed by subgoals or special parameters. It is also true in Program (b) which has a parameter link formed by the subgoal *modify(X,Y)* in the rule *g2*. Since a complicated situation can result from using different types of parameter links, we need to develop a systematic approach to detect and analyze the recursive definition with different kinds of parameter links. This will be discussed in Section 3.3 in detail. In general, we conclude that a parameter link is formed between two same-name parameters in the recursive rule head and in the recursive subgoal only if it can pass data to other parameter links formed by subgoals or special parameters.

3.1.6 The Significance of Parameter Links

In conclusion, the relationship between a parameter link and a parameter modifying process is clear. A parameter link is an indicator of all the processes that involve passing and modifying some values from some parameters in the head to some parameters in the recursive subgoal of one level of recursion. Although a parameter link can always ensure that some values can pass through at least one level of recursion, it cannot guarantee that the value passed to the next level of recursion will always be different from the value received from the previous level. This is particularly true in the case where the parameter link is formed by subgoals. In Figure 3.4, we originally have a parameter link that can change and pass values in different levels of recursion. However, a little change in the procedure defining the subgoal *modify(X,A)* can take away the modification ability of the parameter link. The parameter link no longer transfers a different value if the procedure *modify* is alternated as:

modify(1,1). modify(2,2). modify(3,3).

Therefore, parameter links alone are too weak for detecting parameter modifying processes in a recursive definition.

However, we may see the significance of parameter links with respect to aspect (2) of the parameter modifying process. With parameter links, we have a tool to identify all the possible channels in a recursive definition for passing (but not modifying) data through one level of recursion. As shown above, the ability for passing data is a necessary (but not sufficient) condition for the presence of aspect (2) of the parameter modifying process and consequently a necessary condition for the existence of any parameter modifying process. Thus, the detection of parameter links in a recursive definition indicates a potential presence of aspect (2) in a recursive definition. There are two implications:

(1) If no parameter link can be found in a recursive definition, one can be sure that it is a nonterminating recursive definition[1].

(2) If there exist some parameter links, further steps should be taken to test for the presence of a parameter modifying process.

Therefore, as indicated in aspect (1), to construct parameter links in a recursive definition provides a preliminary test for nontermination in Prolog. Furthermore, it is a test without using any semantic knowledge as the detection of parameter links does not require any semantic knowledge. It constitutes the first step of our non-tracing semantic-knowledge-free pure Prolog nontermination detection technique. Parameter links are thus important in providing a basis for the further steps in nontermination detection. After identifying the parameter links, one can isolate all the possible candidates for the further detection of the parameter modifying process in a recursive definition.

---

1. A recursive definition without any parameter link usually run into endless recursion once the recursive definition is evaluated. However, such a recursive definition can sometimes be prevented from nontermination because no recursion can be invoked under some situation. For example, if we supply the query "?- g(1,X)" for the following recursive definition, no nontermination occurs although nontermination will be the result if the query "?- g(X,Y)" or "?-g(2,X)" is supplied.

```
g(X, Y)  :-  a(X, A),  g(P, Q).
a(2, 3).
```

3.2 Cyclic Parameter Link

However, the presence of a parameter link cannot guarantee that data can be passed through other successive levels of recursion after the first one. The example in Figure 3.8 shows how a recursive definition can have no parameter modifying process even though some parameter links are present. If the query *?- g(X, Y, Z)* is given for the evaluation, the value of *2* will always be passed onto the next level of recursion through the third parameter of the recursive subgoal. But the value is also lost in every level of recursion. Although the parameter link between parameters *X* and *C* can transfer data through one level of recursion, it does not pass data continuously throughout every level of recursion. This kind of parameter link cannot really establish aspect (2) of the parameter modifying process. Although it can pass some values to the next level of recursion, no value is passed on continuously during the recursion. And it does not provide a means to modify the data throughout the recursion. The search tree in Figure 3.8 clearly shows that the third parameter of the recursive subgoal is always instantiated to the same value in an endless recursion. It is obvious that an exit condition can never be reached in this situation. In fact, to have data passed through the successive levels of recursion continuously, a recursive definition must include not only a parameter link but also a cyclic parameter link.

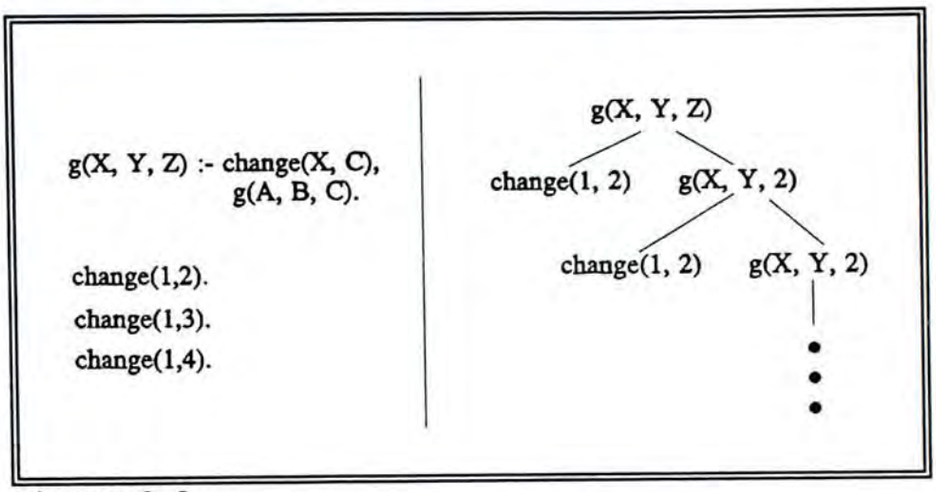


Figure 3.8

A cyclic parameter link is said to exist in a recursive definition when:

(1) there is a dependent relationship extending over all levels of recursion, and

(2) there are two parameters in two different levels of recursion, and

(3) these two parameters occupy the same position in the parameter list of the recursive definition.

The parameter list of a direct recursive definition refers to the parameters used in the recursive subgoal and the parameters used in the head of the recursive rule. For example, the recursive definition below has a parameter list of **n** parameters.

$$g(X_1, X_2, ..., X_n) :- \ldots, \; g(Y_1, Y_2, ..., Y_n), \ldots \; .$$

A cyclic parameter link is formed if there is a dependent relationship between X_i and Y_i, or there are dependent relationships between X_i and Y_j and between X_j and Y_i. In the following paragraphs, we shall show how a cyclic parameter link can be constructed out of parameter links. We shall also explain how a cyclic parameter link can ensure that the result of modification will be passed onto the different levels of recursion. In fact, a cyclic parameter link represents a dependent relation extending over all levels of recursion. Then the difference between the cyclic parameter link in the direct recursive definition and the cyclic parameter link in the indirect recursive definition will be discussed.

Before we begin our discussion on cyclic parameter link, we must first look at how data are transferred in Prolog programs. Two different situations will need to be considered separately:

(1) passing data **from one subgoal to another subgoal** within the same rule, and

(2) passing data **from one rule to another rule.**

There are two ways to accomplish data passing: (1) Data are passed by using parameters with the **same name**, and (2) data are passed by using the parameters at the **same position in the parameter list** of a recursive definition. In situation (1), data are transferred through parameters with the same name. We have seen many examples in the discussion of parameter links. In situation (2), parameters at the same position in the parameter list are used to pass data between different rules. Situation (2) happens when a subgoal in a rule is defined by a procedure that includes some rules. During the evaluation of this rule, this particular subgoal needs to be solved, so the unification

process as described in Figure 2.1 is invoked to solve it. This subgoal will be unified with the head of the rule defining this subgoal. Therefore the parameters in the parameter list of the subgoal are unified with the parameters at corresponding positions in the parameter list of the head of the selected rule. If any parameter in the subgoal in the original rule has already been instantiated to any value, the parameter at the same position in the parameter list of the head of the selected rule now has also the same value. From another point of view, the value in the subgoal of a particular rule can be considered to be transferred to the head of another rule through the two parameters at the same parameter list position of the subgoal in the original rule and the head of the selected rule.

To establish aspect (2) of the parameter modifying process in a recursive definition, the recursive definition must have a parameter link to permit data to be transferred between successive levels of recursion. Therefore, when one considers how to establish aspect (2) of the parameter modifying process, he is actually considering how to pass data along parameter links during recursion. If we examine the evaluation of a recursive definition, we can see that passing data from one level of recursion to the next is equivalent to passing data from one rule to another rule (although the another rule is equivalent to the original one in a direct recursive definition). Therefore, passing data between different levels of recursion is accomplished by parameters at the same position in the parameter list of the recursive definition. To simplify our discussion, we can say that two parameter links are connected to each other if the following situation happens:

> In a certain rule, a parameter link is established between a parameter in the rule head and a particular parameter in a certain subgoal in the body. This subgoal invokes another rule which also has a parameter link and this parameter link extends from a parameter in the rule head. This parameter in the rule head has the same parameter list position as the parameter in the subgoal of the original rule.

So, to establish aspect (2) of the parameter modifying process in a recursive definition, at least one of the parameter links in each level of recursion must always be linked to a parameter link in a previous level of recursion and to a parameter link in the next

level. However, when parameter links in different levels of recursion are linked together, the parameters involved in these parameter links are all in a dependent relationship. In other words, a dependent relationship that extends over different levels of recursion has been established.

Through the parameters that have the same parameter list position, a parameter link in one recursion level can be further connected to another parameter link in the next recursion level. However, the presence of aspect (2) of the parameter modifying process requires that these parameter links are linked up not only for a limited number of recursion levels but for all levels of recursion. In other words, the parameter links must be linked up indefinitely throughout all the recursion process. **Otherwise, the recursive definition given is an improper recursive definition.** The reason is obvious: If parameter links in one level of recursion cannot be connected to the parameter links in the next level, the result of modification will be lost in the next level. Usually, nontermination will happen. However, **a more serious problem is that such a recursive definition is meaningless.** Recursion goes on without performing any real data processing. To have certain parameter links in a recursive definition to be linked up throughout all recursion levels, each parameter link must in some way be linked up to itself again after a certain number of recursion levels. Since the number of parameters in the parameter list of a recursive definition is limited, the number of parameter links that can be formed is also limited. **To have a limited number of parameter links to be linked up throughout an indefinitely number of recursion levels, some of them must be repeatedly used in the process.** In Figure 3.9, **Program (a)** and (b) both have parameter links that extend over more than one level of recursion. But the parameter links in Program (a) can extend over only two levels of recursion while those in Program (b) can extend indefinitely. The difference between them is that the parameter links in Program (b) can be linked up to themselves after two levels of recursion. To have a parameter link linked up to itself after several levels of recursion,

Program (a)	Program (b)
a(X, Y, Z) :- g1(X, B), g2(Y, C), a(A, B, C).	a(X, Y, Z) :- g1(X, B), g2(Y, A), a(A, B, C).

Figure 3.9

there must be a dependent relationship extending over several levels of recursion with two parameters in different levels but at the same position in the parameter list. In other words, there is a cyclic parameter link in the recursive definition. Thus aspect (2) of the parameter modifying process requires the presence of a cyclic parameter link.

There are two different situations that require a different consideration in establishing cyclic parameter links. They are situations in which:

(1) only one parameter link exists in the recursive definition, or

(2) more than one parameter link exists in the recursive definition.

If there is only one parameter link in a recursive definition, it is easy to determine whether this parameter link is cyclic or not. A parameter link can be a cyclic parameter link only if the position of the parameter in the rule head involved in the parameter link is equal to the position of the parameter in the recursive subgoal involved in the parameter link. If there is only one parameter link in the recursive definition, the parameter link must be linked up to itself in the immediately next level of recursion. Otherwise, the data passed onto the next level will be lost. The difference between the recursive definitions with and without a cyclic parameter link can be illustrated in Figure 3.10. In the search trees given, we can see how data are modified along a cyclic

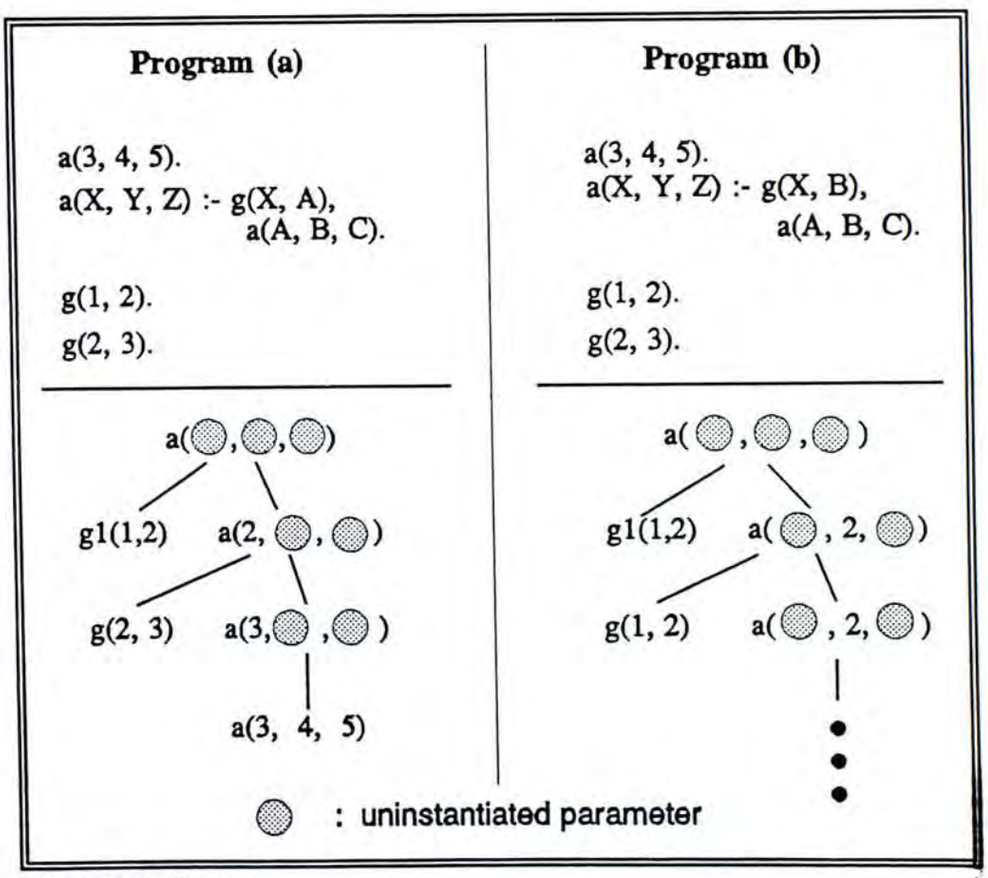


Figure 3.10

parameter link, and on the other hand how the result of modification is lost in the recursive definition if there is no cyclic parameter link. When more than one parameter link exist in a recursive definition, there are different ways in which these parameter links can be linked up to form a cyclic parameter link. The different examples in Figure 3.11 show how different cyclic parameter links can be formed in a recursive definition with multiple parameter links. In general, a cyclic parameter link is formed by either linking up a parameter link to itself directly or indirectly. For a recursive definition with only one parameter link as what is shown by Program (a) in Figure 3.10 or Program (c) in Figure 3.11, a cyclic parameter is formed by linking up the parameter to itself directly. In Examples (a) and (b) of Figure 3.11, the cyclic parameter link is formed completely by indirect linking-up.

Program (a)	Program (b)	Program (c)
a(X, Y, Z) :- g1(X, B), g2(Y, C), g3(A, Z), a(A, B, C).	a(X, Y, Z) :- g1(X, B), g2(Y, A), a(A, B, C).	a(X, Y, Z) :- g1(X, A), g2(Y, C), a(A, B, C).

Figure 3.11

To establish that a cyclic parameter link exists in an indirect recursive definition, it is necessary to establish some ordinary parameter links in the indirect recursive definition first. As described in Section 3.1, the parameter link of an indirect recursive definition can be established by linking up parameter links in every rule involved in the definition. Then a cyclic parameter link can be formed if certain parameter links of the recursive definition can somehow be connected to themselves. It is similar to the case of direct recursive definitions.

If there is a cyclic parameter link in a recursive definition, we can say that this recursive definition has a potential parameter modifying process since a cyclic parameter link guarantees that:

(1) there is a process in which the values in some parameters can be modified in each level of recursion, and

(2) the result of modification can be passed from the recursion level, and the modified result can be passed onto the next level.

The effect of a cyclic parameter link is equal to a parameter modifying process if one of the parameters involved in the cyclic parameter link can be modified during the recursion. On the other hand, if no cyclic parameter link exists in a recursive definition, the implication is that no potential parameter modifying process exists, and thus a parameter modifying process does not exist either. Therefore, parameter analysis provides a means to detect the nontermination caused by the absence of a parameter modifying process, and it does not require any semantic knowledge. Moreover, parameter analysis is important in the whole process of nontermination detection as it also identifies any potential parameter modifying process. The next step is to verify whether these potential parameter modifying processes (which are also potential exit-variable modifying processes), can act as an exit-reaching process. This requires the detection of the exit condition in a Prolog program, which in turn implies the need for semantic knowledge of the program. However, we can show in Chapter 4 that this is not necessarily the case as far as pure Prolog is concerned. The technique to be presented in Chapter 4 and parameter analysis together form a diagnostic test for nontermination in pure Prolog programs without the need of semantic knowledge. Parameter analysis is an important part in the test as it supplies the necessary data--the potential parameter modifying processes, (i.e., the cyclic parameter links)--for further examination.

In the following sections, we shall show how parameter analysis can be achieved by incorporating the graph technique in the cyclic parameter link detection. In the methods below, no special handling is provided for the improper recursive definition mentioned in Section 3.2. If in a recursive definition there are several parameter links that can be linked up together across several levels of recursion but they do not form a cyclic parameter link, this recursive definition will be treated like any other recursive definitions having no cyclic parameter link.

3.3 Parameter Link Detection

In this section, we shall describe how a systematic approach can be developed to detect a parameter link and in turn a cyclic parameter link. Essentially, our method is based on the graph technique. After showing how to use the graph technique to detect parameter links and cyclic parameter links, algorithms will be developed to illustrate the method of parameter analysis.

3.3.1 Graph Technique

3.3.1.1 Preliminaries

To establish a graph technique to detect parameter links and cyclic parameter links, we must first develop some notations to represent the recursive definition being analyzed.

In each rule, the parameters in the rule head and the parameters in the recursive subgoal are represented by circles with a number inside. The number within the circle indicates the position of this parameter at the parameter list. Those parameters in the same parameter list are arranged in the same row enclosed by a pair of parentheses with a predicate name at the leftmost position. The example below will have a graphical representation for its parameters as the one in Figure 3.12.

a(X,Y,[Z|C]) :- g(X,B), h(Y,D), a(X,B,C).

The upper row of circles in Figure 3.12 indicates the parameters from the head of the rule *a*. The row representing the parameters from the rule head is always placed above the row representing the parameters from the subgoal.

If a parameter link exists between a parameter in the rule head and the parameter in the recursive subgoal, and this parameter link is not formed by the

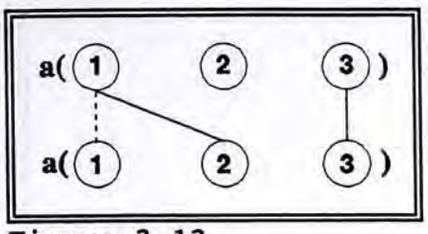


Figure 3.13

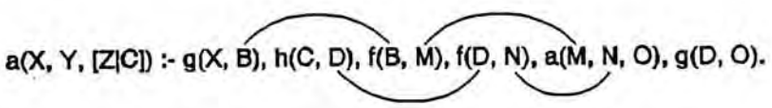
parameters with the same name, a solid line is drawn between the two circles representing these two parameters. If it is formed by the parameters with the same name, a broken line is drawn instead. So the above rule will have a graphical representation as in Figure 3.13.

3.3.1.2 on Parameter Links

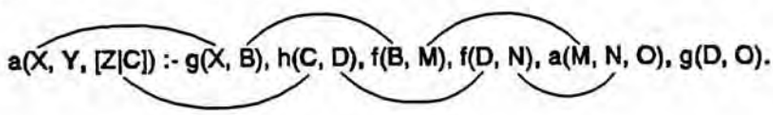
To detect a parameter link in a direct recursive definition, we need to first find out all the parameters linked to the parameter in the head of the recursive rule. Then we check whether some of these parameters are also present in the parameter list of the recursive subgoal. For the parameters within the same subgoal, they are already linked together since they are dependent on each other. However, among the different subgoals, the parameters are linked only if they can pass data from one parameter to another. Hence two parameters in different subgoals are linked together only if they have the same name. For each pair of parameters with the same name but in different subgoals, an arc is placed above them in the rule. Only those subgoals preceding the recursive subgoal are considered because those subgoals after the recursive subgoal cannot have effect on the data being passed onto the next level of recursion. Therefore the rule below:

a(X,Y,[Z|C]) :- g(X,B), h(C,D), f(B,M), f(D,N) a(M,N,O), g(D,O).

will have arcs over it like this:



Now, by adding
more arcs to join the parameters in the body and the parameters in the head that have the same name, we can find out which parameter in the head forms a parameter link to the parameter in the recursive subgoal. The final result for the above example is:



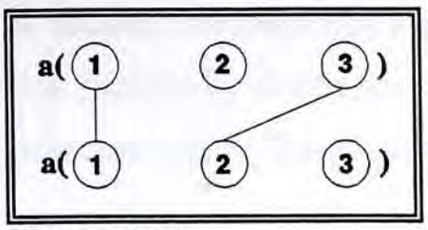


Figure 3.14

Parameter links are found between parameters in the head and the parameters in the recursive subgoal at the following positions: first to first and third to second. The graphical representation for the parameter links in this recursive definition can be drawn as Figure 3.14.

In an indirect recursive definition, parameter links in each rule must first be constructed and drawn in a manner similar to above. After a graphical representation for each rule involved in this indirect recursive definition has been drawn, we go on to check whether the parameter links in these rules can be linked up together to form a parameter link for the entire recursive definition. This can be accomplished by using graphical representation as well. The graphical representations of the rules are examined. If any rule has the upper row identical to the lower row of another rule (two rows are identical if they have the same predicate name and same number of circles), these two graphical representations are merged together as is illustrated in Figure 3.15 and Figure 3.16. A parameter link for the entire recursive definition is said to be present if a line can be drawn from the first row to the second row. Attention must be paid to handling different types of parameter links. When a broken line is linked to a solid line, the broken line becomes a solid line. Otherwise, the broken line remains as a broken line.

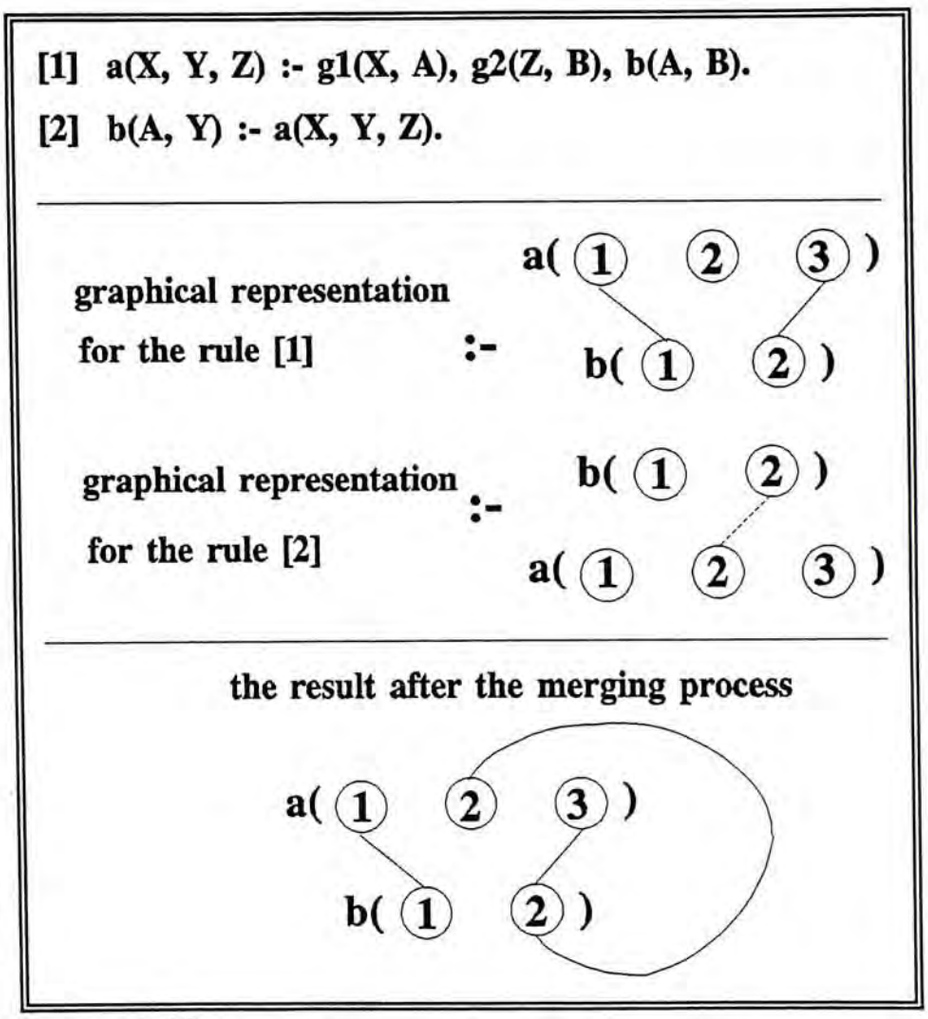


Figure 3.15

A broken line becomes a solid one in this case because it indicates that the parameter link formed by the parameters with the same name (represented by the broken line) is now responsible for passing some data and plays a significant role in the parameter link.

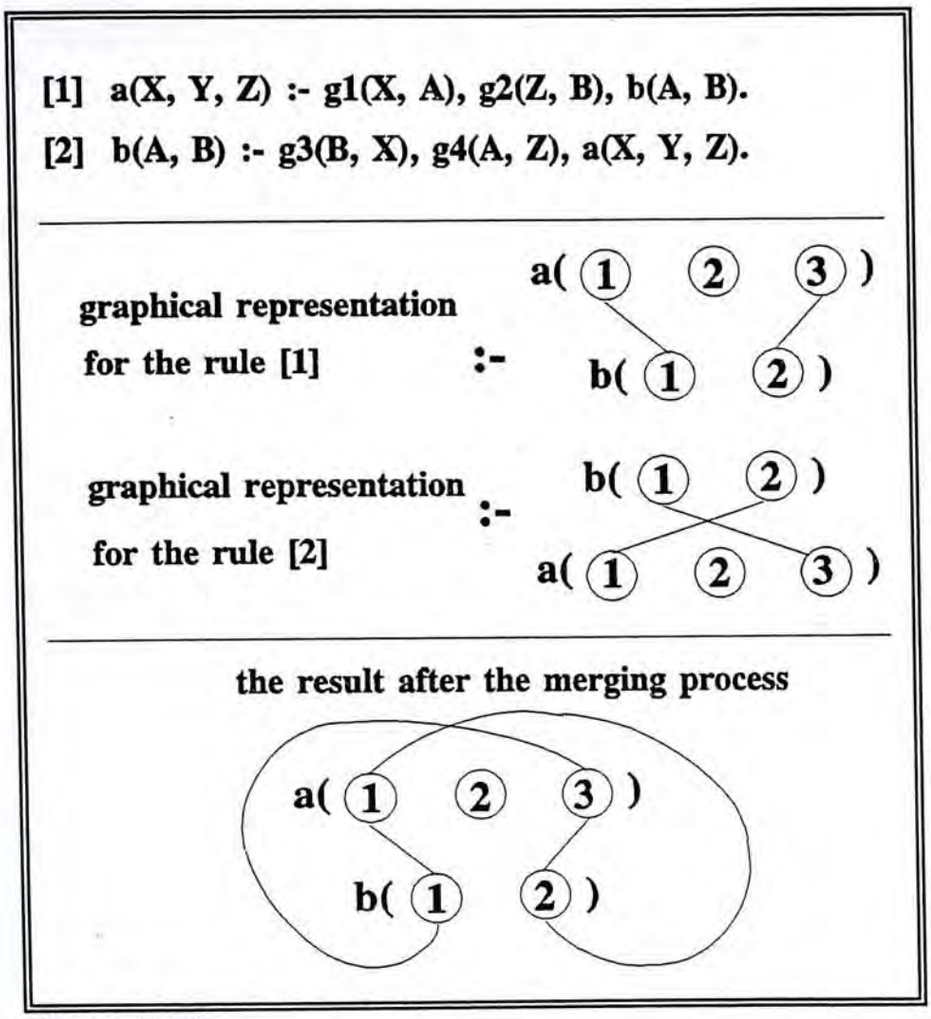


Figure 3.16

Comparing the results of these two examples, we can clearly see the difference between them. In Figure 3.16, the final graphical representation shows that a cycle is formed among its parameters involved in the parameter links while no cycle appears in the graphical representation of the example in Figure 3.15. Actually, only the example in Figure 3.16 has a cyclic parameter link. This is the reason for the name, *cyclic parameter link*: the link appears as a cycle in the graph. This merging process can also be applied to the graph obtained from a direct recursive definition.

In Figure 3.14, the graphical representation is now merged and becomes the picture in Figure 3.17. In this case, a cycle also appears at the first parameter of the recursive definition.

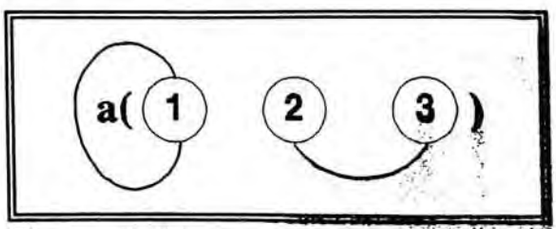


Figure 3.17

3.3.2 Algorithms

The first algorithm to be presented in this section identifies a parameter link within one rule. In order to improve the readability of the algorithms, comments enclosed between "[ ]" are given in the algorithms.

Algorithm 3.1 Constructing parameter links in one rule

INPUT : a rule, $\mathbf{g}(A_1,...,B_n)$:- $\mathbf{sg}_1(...)$, ..., $\mathbf{sg}_k(B_1,...,B_m)$, ..., $\mathbf{sg}_i(...)$. and the position of the recursive subgoal $\mathbf{sg}_k$, **k**. (In a direct recursive definition, $\mathbf{sg}_k(...) = \mathbf{g}(...)$.)

OUTPUT : **PS**, a set of sets {(**X**,**Y**,**Z**), **s**, **p**}. The triple indicates a parameter link between the recursive rule head and the recursive subgoal: **X** is a parameter position in rule head, **Z** is a parameter positions in the recursive subgoal and **Y** shows the parameter type. **s** is a set of subgoals that form the parameter link; **p** is a set of parameters involved in the parameter link.

```
[1]   S  := { {sg1(X1, ..., Xa)} }          [ S is a set of subgoal sets ]
[2]   P := { {X1, ..., Xa} }                [ P is a set of parameter sets ]
[3]   PS := {}

[4]   For j := 2 to k - 1 Do
      Begin                                  [ to form subgoal sets and parameter sets ]
[5]      count := 0
[6]      For each subset si in S Do
[7]         If any parameter in sgj(X1, ..., Xb) appears in pi in P  Or  any parameter in sgj(X1, ..., Xb)
                 is related to any parameter in pi through special parameters
               Then  Begin
[8]                     count := count + 1
[9]                     si := si  ∪  { sgj(X1, ..., Xb) }
[10]                    pi := pi  ∪  { X1, ..., Xb }
                     End
[11]     If count = 0
            Then  Begin
[12]                  S := S  ∪  { { sgj(X1, ..., Xb) } }
[13]                  P := P  ∪  { { X1, ..., Xb } }
                   End
[14]  End
```

Algorithm 3.1 (continued)

```
[15]  For each subset p_i in P Do                    [ to form all possible parameter links ]
      Begin
[16]    If any parameter in p_i is also in set p_j, p_j ∈ P Or
          any parameter in p_i is related to a parameter in p_j through special parameters
[17]    Then Begin
                 s_temp := s_i ∪ s_j
[18]             S     := S - { s_i } - { s_j }
[19]             S     := S ∪ { s_temp }
[20]             p_temp := p_i ∪ p_j
[21]             P     := P - { p_i } - { p_j }
[22]             P     := P ∪ { p_temp }
               End
      End

[23]  For a := 1 to n Do                             [ to form parameter links between ]
      Begin                                          [ parameters in the recursive rule ]
[24]    For each set p_i in P Do                     [ head and the recursive subgoal ]
[25]      If parameter A_a in the rule head g also appears in p_i Or
            A_a is related to a parameter in p_i through special parameters
[26]        Then For b := 1 to m Do
[27]               If B_b in the recursive subgoal sg_k also appears in p_i Or
                    B_b is related to a parameter in p_i through special parameters
[28]                 Then PS := PS ∪ { {(a,1,b), s_i, p_i} }     [ "1" is for a normal ]
      End                                                      [ parameter link ]

[29]  For i := 1 to n Do                                       [ "0" is for a same-name ]
[30]    For j := 1 to m Do                                     [ parameter link ]
[31]      If parameter A_i in g = parameter B_j in sg_k
[32]        Then PS := PS ∪ { {(i,0,j), {}, {}} }

[33]  For i := 1 to n Do
[34]    For j := 1 to m Do
[35]      If parameter A_i in g is directly related to parameter B_j in sg_k through special parameters
[36]        Then PS := PS ∪ { {(i,1,j), {}, {A_i, B_j}} }
```

We can show how this algorithm works by considering that the following rule supplied as input:

a(X,Y,[Z|L]) :- g1(A,C), g2(M,N), g3(A,L), g4(X,M), g5(N,B), b(A,B,C,Y).

where b(A,B,C,Y) is the recursive subgoal. In lines [1] and [2] of the algorithm, the subgoal set **S** is initialized to be *{{g1(A,C)}}* while the parameter set **P** is initialized to be *{{A,C}}*. Because there are five subgoals preceding the recursive subgoal, line [4] to line [14] will be repeated four times. In the first iteration, because the parameters in *g1(A,C)* do not appear in the parameters of the subgoal *g2*, the variable **count** remained as zero and the *If* statement from line [11] to line [14] is performed. **S** becomes *{{g1(A,C)}, {g2(M,N)}}* and **P** becomes *{{A,C}, {M,N}}*. In the second iteration, because parameter *A* in the subgoal *g3(A,L)* also appears in the set *{g1(A,C)}* of the subgoal set **S**, the *If* statement from lines [7] through [10] can be activated so that *{g1(A,C)}* is modified to be *{g1(A,C),g3(A,L)}* and *{A,C}* in **P** is modified to be *{A,C,L}* after the *For* loop from line [6] to line [10] is completed. The subgoal set **S** becomes *{{g1(A,C),g3(A,L)}, {g2(M,N)}}* while **P** becomes *{{A,C,L}, {M,N}}*. After exiting the first nested *For* loop, the subgoal set **S** becomes *{{g1(A,C), g3(A,L)}, {g2(M,N), g4(X,M), g5(N,B)}}* and the parameter set **P** becomes {{A,C,L}, {M,N,X,B}}. Since the two sets in **P** do not have any parameter in common, lines [15] through [22] of the algorithm do not have any effect on **S** and **P**. From line [23] to line [28], the set **PS** is formed. Because there are three parameters in the recursive rule head, lines [23] through [28] are repeated three times. In the first iteration, the parameters in every subset in **P** are compared to the first parameter in the recursive rule head, *X*, and all the parameters in the recursive subgoal, *A,B,C,Y*. Since *X*, the first parameter in the recursive rule head, and *B*, the parameter in the recursive subgoal, both appear in the second subset of **P** *{M,N,X,B}*, the set **PS** is changed from *{}* to *{ { (1,1,2), {g2(M,N),g4(X,M),g5(N,B)}, {M,N,X,B} } }* after the first iteration. After the three iterations, the resulting **PS** becomes *{ {(1,1,2), {g2(M,N), g4(X,M), g5(N,B)}, {M,N,X,B}}, {(3,1,1), {g1(A,C),g3(A,L)}, {A,C,L}}, {(3,1,3), {g1(A,C),g3(A,L)}, {A,C,L}} }*. Finally, lines [29] through [32] expand **PS** to be *{ {(1,1,2), {g2(M,N),g4(X,M),g5(N,B)}, {M,N,X,B}}, {(3,1,1), {g1(A,C),g3(A,L)}, {A,C,L}}, {(3,1,3), {g1(A,C),g3(A,L)}, {A,C,L}}, {(2,0,4), {}, {}} }*. Because no parameter in recursive rule head can relate to any parameter in recursive subgoal through special parameters, lines [33] through [36] do not affect the output **PS**. The result indicates that three proper parameter links exist between the parameters in the rule head and the parameters in the recursive subgoal with the following positions: third to first, third to

third and first to second. Moreover, there is a parameter link formed by the parameters with the same name in the position: second to fourth.

The next algorithm shown below constructs parameter links for the entire recursive definition with different rules involved in an indirect recursive definition. If it is a direct recursive definition, the result of Algorithm 3.1 indicates the parameter links occurring in one level of recursion.

Algorithm 3.2 Constructing parameter links for an indirect recursive definition

INPUT : a sequence of sets of triples PS_1, PS_2, ..., PS_k; each set is for one rule. (If only a direct recursive definition is examined, **k** = **1**; otherwise, **k** is a finite number which is greater than **1** if an indirect recursive definition is involved.) They are arranged in an order the same as the order of evaluation.

OUTPUT : the set **PS'** which contains some triples indicating the positions of the parameters involved in a parameter link of an indirect recursive definition.

```
[1]   PS'        := {}
[2]   PS_temp    := {}
[3]   s_temp     := {}
[4]   p_temp     := {}
[5]   i          := 1

[6]   While i < k and PS_i ≠ {} Do
      Begin
[7]     For each set in PS_i, {(X_m,Y_m,Z_m), s_m, p_m}, Do
[8]       For each set in PS_i+1, {(X_n,Y_n,Z_n), s_n, p_n}, Do
[9]         If Z_m = X_n
[10]          Then If Y_m = 1 Or Y_n = 1
                   Then Begin
[11]                    If i = 1 Then Begin
[12]                                    s_temp := {s_m,s_n}
[13]                                    p_temp := {p_m,p_n}
                                      End
                                 Else Begin
[14]                                    s_temp := s_temp ∪ {s_n}
[15]                                    p_temp := p_temp ∪ {p_n}
                                      End
[16]                    PS_temp := PS_temp ∪ { {(X_m, 1, Z_n), s_temp, p_temp} }
                   End
[17]          Else PS_temp := PS_temp ∪ { {(X_m, 0, Z_n), {}, {}} }
```

Algorithm 3.2 (continued)

[18] **i** := **i** + 1
[19] $\mathbf{PS}_i := \mathbf{PS}_{temp}$
End

[20] **PS'** := $\mathbf{PS}_k$

Consider the following example indirect recursive definition involving three rules:

```
a(X,Y,[Z|L]) :- g1(A,C), g2(M,N), g3(A,L), g4(X,M), g5(N,B), b(A,B,C).
b(X, Y, Z)   :- f1(Y, D), f2(Z, A), c(A, B, C).
c(X, Y, Z)   :- g1(X, C), g2(Y, B), a(A, B, C).
```

By applying Algorithm 3.1 to every rule, $\mathbf{PS}_1$ of the first rule is *{ {(3,1,1), {g1(A,C),g3(A,L)}, {A,C,L}}, {(3,1,3), {g1(A,C),g3(A,L)}, {A,C,L}}, {(1,1,2), {g2(M,N), g4(X,M), g5(N,B)}, {M,N,X,B}}, {(2,0,4), {}, {}} }*, $\mathbf{PS}_2$ of the second rule is *{ {(2,1,4), {f1(Y,D)}}, {(3,1,1), {f2(Z,A)}} }* and $\mathbf{PS}_3$ of the third rule is *{ {(1,1,3), {g1(X,C)}, {X,C}}, {(2,1,2), {g2(Y,B)}, {Y,B}} }*. In this case, the constant, **k**, in line [6] is **3** so that only two iterations can occur. After the first iteration generated by the *While* loop in lines [6] through [19], $\mathbf{PS}_{temp}$ becomes *{ {(3,1,1), {{g1(A,C), g3(A,L)}, {f2(Z,A)}}, {{A,C,L}, {Z,A}} }, {(1,1,4), {{g2(M,N), g4(X,M), g5(N,B)}, {f1(Y,D)}}, {{M,N,X,B}, {Y,D}} } }* because of the triples *(3,1,3)* and *(1,1,2)* in $\mathbf{PS}_1$ and *(2,1,4)*, *(3,1,1)* in $\mathbf{PS}_2$. In line [19], $\mathbf{PS}_{temp}$ is assigned to $\mathbf{PS}_2$ in the first iteration. Therefore, the triples in $\mathbf{PS}_2$ for the second iteration are *(3,1,1)* and *(1,1,4)*. Since the original $\mathbf{PS}_3$ has the triples *(1,1,3)* and *(2,1,2)*, $\mathbf{PS}_{temp}$ becomes *{ {(3,1,3), {{g1(A,C), g3(A,L)}, {f2(Z,A)}, {g1(X,C)}}, {{A,C,L}, {Z,A}, {X,C}} } }* because only *(3,1,1)* in $\mathbf{PS}_2$ and *(1,1,3)* in the original $\mathbf{PS}_3$ can satisfy the condition $Z_m = X_n$ in line [9]. After the *While* loop, $\mathbf{PS}_3$ becomes *{ {(3,1,3), {{g1(A,C), g3(A,L)}, {f2(Z,A)}, {g1(X,C)}}, {{A,C,L}, {Z,A}, {X,C}} } }* too. Eventually, the set **PS'** attains the same set of elements as the set $\mathbf{PS}_3$ because of line [20].

Finally, the complete algorithm for parameter analysis is presented as follows:

Algorithm 3.3 A parameter analysis algorithm

INPUT : a recursive definition

OUTPUT : **CPS**, a set of cyclic parameter links, and **PS'**, a set of parameter link sets.

apply Algorithm 3.1 to each recursive rule in the recursive definition to generate PS_1, PS_2, ..., PS_k
[**k** = 1 for a direct recursive definition]

If only one rule is involved in the recursive definition [i.e., it is a direct]
Then $PS_{new} := PS_1$ [recursive definition]
$PS' := \{ PS_1 \}$ [PS is a parameter link set]
Else apply Algorithm 3.2 with PS_1, PS_2, ..., PS_k
$PS_{new} := PS'$

[1] $PS_{temp} := \{\}$
[2] $CPS := \{\}$
[3] $CPS' := \{\}$
[4] $PS_{old} := PS_{new}$
[5] $k := 1$

[6] While $k < n$ and $PS_{new} \neq \{\}$ Do [n = arity of the recursive rule head]
Begin
[7] For each set $\{(X_i, Y_i, Z_i), s_i, p_i\}$ in PS_{new} Do
[8] For each set $\{(X_j, Y_j, Z_j), s_j, p_j\}$ in PS_{old} Do
[9] If $Z_i = X_j$
[10] Then If $Y_i = 1$ Or $Y_j = 1$
Then Begin
[11] If $k = 1$
Then Begin
[12] $ss_{temp} := \{s_i, s_j\}$
[13] $sp_{temp} := \{p_i, p_j\}$
End
Else Begin
[14] $ss_{temp} := ss_{temp} \cup \{s_j\}$
[15] $sp_{temp} := sp_{temp} \cup \{p_j\}$
End
[16] $PS_{temp} := PS_{temp} \cup \{\{(X_i,1,Z_j),ss_{temp},sp_{temp}\}\}$
End
[17] Else $PS_{temp} := PS_{temp} \cup \{ \{(X_i, 0, Z_j), \{\}, \{\}\} \}$
[18] For each set $\{(X_i, Y_i, Z_i), ss_i, sp_i\}$ in PS_{temp} Do [to identify the cyclic parameter links]
[19] If $X_i = Z_i$ Then Begin
[20] $CPS' := CPS' \cup \{ \{(X_i, Y_i, Z_i), ss_i, sp_i\} \}$
[21] $PS_{temp} := PS_{temp} - \{ \{(X_i, Y_i, Z_i), ss_i, sp_i\} \}$
End

Algorithm 3.3 (continued)

```
[22]    PS_new := PS_new ∪ PS_temp
[23]    PS_temp := {}
[24]    k := k + 1
      End

[25]  For each set {(X_i, Y_i, Z_i), ss_i, sp_i} in CPS' Do
[26]    If Y_i <> 0
[27]      Then CPS := CPS ∪ { {X_i, ss_i, sp_i} }
```

Consider the following simplified example. In the following discussion, only the positions of the parameters involved are mentioned to give a clearer picture of how the method of parameter analysis can detect cyclic parameter links. Suppose that the application of Algorithm 3.2 to a certain recursive definition results in a $\mathbf{PS}_{new}$ of *{ {(3,1,3), ...}, {(1,0,2), ...}, {(5,1,2), ...}, {(2,1,4), ...}, {(2,1,5), ...} }*. Lines [1], [2] and [3] initialize all the sets $\mathbf{PS}_{temp}$, **CPS** and **CPS'** to be empty sets while line [4] initialize the sets $\mathbf{PS}_{new}$ and $\mathbf{PS}_{old}$ to contain the same set of elements. The number of iterations **n** in line [6] is obtained by counting the number of parameters in the recursive rule head. Suppose that there are five parameters in the recursive rule head (in our example, the number of parameters cannot be less than 5 since **PS'** contains *(5,1,2)* and *(2,1,5)*), the *While* loop located between line [6] and line [24] can repeat five times. In the first iteration, $\mathbf{PS}_{temp}$ is first changed from *{}* to *{ {(3,1,3), ...}, {(1,1,4), ...}, {(1,1,5), ...}, {(5,1,4), ...}, {(5,1,5), ...}, {(2,1,2), ...} }* after line [17]. However, the set $\mathbf{PS}_{temp}$ is further modified in lines [18] through [21]. In the *For* loop located in lines [18] through [21], all cyclic parameter links in $\mathbf{PS}_{temp}$ are taken away and added to the set **CPS'**. Therefore, the resulting $\mathbf{PS}_{temp}$ becomes *{ {(1,1,4), ...}, {(1,1,5), ...}, {(5,1,4), ...} }* while **CPS'** becomes *{ {(3,1,3), ...}, {(5,1,5), ...}, {(2,1,2), ...} }*. Then, in line [22], the set $\mathbf{PS}_{temp}$ and the set $\mathbf{PS}_{new}$ is merged together to form a new $\mathbf{PS}_{new}$ for the next iteration. Consequently, the set $\mathbf{PS}_{new}$ for the second iteration becomes *{ {(1,1,4), ...}, {(1,1,5), ...}, {(5,1,4), ...}, {(3,1,3), ...}, {(1,0,2), ...}, {(5,1,2), ...}, {(2,1,4), ...}, {(2,1,5), ...} }*. After five iterations of the *While* loop, the set **CPS'** becomes *{ {(3,1,3), ...}, {(5,1,5), ...}, {(2,1,2), ...} }*. Finally, the set of cyclic parameter links **CPS** is calculated from **CPS'** in

lines [25] through [27] by eliminating the cyclic parameter links established through only same-name parameter. Therefore, only the appropriate cyclic parameter links, i.e., the cyclic parameter links of type "1", will be included in the final set of cyclic parameter links.

In conclusion, if the set of cyclic parameter links, **CPS**, obtained is an empty set, then no cyclic parameter link exists in the recursive definition or the recursive definition is not properly defined. In the former case, nontermination will occur in the evaluation of this recursive definition since no parameter modifying process exists. In the latter case, it is not certain at this stage whether there is nontermination. Otherwise, the elements in **CPS** will indicate the positions of parameters involved in the cyclic parameter link. **PS'** is also provided since any further analysis will require not only the parameters involved in the cyclic parameter link but also each parameter link occurring in this recursive definition.

CHAPTER 4— Data Analysis

The parameter analysis described in the last chapter actually plays a twofold role in detecting nontermination in Prolog programs. On the one hand, the parameter analysis itself is a nontermination detection technique because it can identify all those recursive definitions that do not have any proper parameter modifying process. As shown in the last chapter, a recursive definition without such process will definitely lead to nontermination. Therefore, parameter analysis is an essential preliminary step in nontermination detection. Through checking cyclic parameter links, any **potential** exit-reaching processes can be located in a recursive definition. The next step is to develop a method that can identify the exit-reaching process from these cyclic parameter links. Apparently, this can only be done by finding all the exit conditions present in a recursive definition. However, as pointed out in the last chapter, semantic knowledge is required to locate an exit condition and this is inconsistent with the goal of this study.

It can be demonstrated that the verification of a potential exit-reaching process **is feasible without having to detect directly any exit condition in a Pure Prolog recursive definition.** In other words, the verification can be done **without resorting to any semantic knowledge in a pure Prolog program.** In parameter analysis, if all recursive definitions are found to have at least one cyclic parameter link, **data analysis** developed in this chapter can then be applied to confirm whether nontermination will really occur. Data analysis involves constructing ***data links*** for these recursive definitions. Essentially this technique relies on **analyzing the data which would pass through the cyclic parameter links.**

Our discussion will be divided into six sections. First, we shall discuss **what a data** link is and how to form it for a recursive definition. Then we **shall examine the** difference between pure Prolog and general Prolog. This will show **us why we should** limit our discussion to pure Prolog. Third, we shall show the **relationship between** data

links and nontermination. Our main concern of how to detect nontermination with the use of data links is discussed in the fourth part of this chapter. By linking up the data links to form a *connected data-link list*, we can detect nontermination by examining the connected data-link list. In the fifth part, a special situation in constructing a data link, namely, the presence of special parameters, will be discussed. Finally, we shall present our method of data analysis with algorithms.

By combining parameter analysis and data analysis, we shall be able to detect the nontermination errors in pure Prolog programs without resorting to any semantic knowledge. This would provide a helpful alternative to the traditional tracing technique in nontermination diagnosis. To prevent our discussions from becoming too confusing, in this chapter, we limit our scope to the case of recursive definitions that have **only one cyclic parameter link** or **several independent cyclic parameter links.** Cyclic parameter links are independent from each other if, for the same recursive definition, the parameters and the subgoals which involve in one cyclic parameter link are entirely different from those which involve in another cyclic parameter link. Then, the recursive rule below has two independent cyclic parameter links:

```
g(X, Y, Z) :- link1(X, A), link2(Y, B), g(A, B, C).
```

while the next recursive rule has two **interdependent** cyclic parameter links:

```
g(X, Y, Z) :- link(X, A,  Y, B), g(A, B, C).
```

The subgoal that forms the first cyclic parameter link between *X* and *A* and the subgoal that forms the second link between *Y* and *B* are in fact the same subgoal, *link*, in the second recursive rule. Moreover, the recursive rule below has two interdependent cyclic parameter links because they share certain common parameters:

```
g(X,Y,Z) :- link1_a(X,C), link1_b(C,A),
            link2_a(Y,C), link2_b(C,B), g(A,B,C).
```

The method developed in this chapter therefore does not handle recursive definitions with multiple interdependent cyclic parameter links. The general treatment of them will be explored in the next chapter.

4.1 Data Links

First, we examine what a data link is and how it can be formed in a recursive definition. **A *data link* exists between any parameter in the head of a recursive rule and any parameter in a recursive subgoal if:**

(1) there already exists a cyclic parameter link between these two parameters, and

(2) each parameter in the cyclic parameter link can be successfully instantiated to a certain value; that is, from the perspective of data transfer, some data can pass through this cyclic parameter link in at least one complete cycle[1].

Because a cyclic parameter link can extend more than one level of recursion, a parameter cycle of recursion can consist of one or more levels of recursion. In order to simplify our discussion, the following discussion will usually be based on the examples with cyclic parameter links extending over only one level of recursion.

In Figure 4.1, two almost identical programs are given. However, only one of them has data links. The contrast between them will help us to understand how one can establish a data link. Program (a) and Program (b) both have the same recursive rule, where subgoals *link1(X,A)* and *link2(A,P)* form a cyclic parameter link between the first parameter in the head, *X*, and the first parameter in the recursive subgoal, *P*. However, only the recursive

Program (a) Recursive definition with a data link		Program (b) Recursive definition without any data link	
goal(X,Y) :- link1(X,A), link2(A,P), goal(P,Q).		goal(X,Y) :- link1(X,A), link2(A,P), goal(P,Q).	
link1(1,2).	link2(3,4).	link1(1,2).	link2(4,5).
link1(2,3).	link2(4,5).	link1(2,3).	

Figure 4.1

1 Because a cyclic parameter link can extend over one or more levels of recursion, one complete cycle of cyclic parameter link can have one or more levels of recursion involved. To facilitate our discussion, since a data link always represents a data transfer through a cyclic parameter link, a data link can be considered to represents a data transfer through a "cyclic parameter link cycle" (or simply "**parameter link**") of recursion. Therefore, *a data link may have one or more levels of recursion involved but still represents a data transfer through one parameter cycle of recursion.*

definition in Program (a) has a data link. There is no data link in the recursive definition in Program (b) where requirement (2) for a data link cannot be met in Program (b) as no data can pass through the cyclic parameter link.

The procedure defining the subgoal *link1* in both programs is formed by two facts, *link1(1,2)* and *link1(2,3)*. Therefore, the first parameter of *link1* can only be instantiated to the values *1* or *2*. However, the first parameter of *link1*, *X*, is one of the parameters in the cyclic parameter link formed by the subgoals *link1* and *link2* and is also the parameter of the rule head, *goal(X,Y)*. Therefore, the parameter *X* regulates the data passed into the cyclic parameter link and the only data that can be passed into this cyclic parameter link are *1* and *2* according to the two facts defining *link1*. If the value *1* is supplied to the cyclic parameter link, according to the recursive definition in Program (b), the parameters of the subgoals in the cyclic parameter link are instantiated as: *link1(1,2), link2(2,P)*. But the subgoal *link2* is defined by a procedure with only one fact, *link2(4,5)*, in Program (b). It is obvious that the term *link2(2,P)* cannot be unified with *link2(4,5)*. Thus unification fails and no data can be successfully passed on. Moreover, if the value *2* is supplied, the parameters of the subgoals will be instantiated as *link1(2,3), link2(3,P)*. The term *link2(3,P)* also fails in any unification. A value that satisfies the subgoal *link1* cannot simultaneously satisfy the subgoal *link2* and vice versa. Therefore, in the concept of the data transfer analogy[1], no data can pass through this cyclic parameter link. Requirement (2) of the data link cannot be fulfilled and no data link exists in Program (b).

On the other hand, a data link can be established in Program (a). No data can pass through the cyclic parameter link when the value *1* is bound to the parameter *X*. But Program (a) is different from Program (b) because the procedure defining the subgoal *link2* is formed by two facts, *link2(3,4)* and *link2(4,5)*, instead of *link2(4,5)* only. When the value *2* is bound to the parameter *X*, the subgoals are instantiated as *link1(2,3), link2(3,P)*. The term *link2(3,P)* can be instantiated as *link2(3,4)* in Program (a) and thus the value *4* can be passed onto the next level of recursion. Requirement

1. Please refer to the discussion in the first paragraph of **Section 3.1 in Chapter 3.**

(2) is thus satisfied and a data link is considered to be present in the cyclic parameter link for Program (a).

The examples in Figure 4.1 shows that **the procedures defining the subgoals in the cyclic parameter link determine the existence of data link** because the procedures defining these subgoals can determine whether requirement (2) for data links can be satisfied. In other words, a data link can only be established if the subgoals in a cyclic parameter link have all been defined by some *appropriate* procedures. An *appropriate* procedure contains at least one fact that can form a data link. Therefore the procedures of subgoals *link1* and *link2* in Program (a) are still appropriate for forming data links if they are reduced to:

link1(2,3). link2(3,4).

Because the fact *link1(1,2)* in the procedure defining the subgoal *link1* and the fact *link2(4,5)* in the procedure of *link2* are not relevant in forming a data link, they can be removed without taking away the data links formed in this recursive definition.

Moreover, data can also pass through a cyclic parameter link that is completely or partially formed by some special parameters, i.e, lists or structured data. Then, apart from subgoals, special parameters can also be used to establish a data link. But the data link formed by a pair of special parameters will have certain special properties that require some complicated analysis. The most significant difference between the data links formed by subgoals and those formed by special parameters is this: while, in the former case, all the subgoals in the data link must be defined by certain procedures, the subgoals forming the data link are not necessarily defined by any procedure in the latter case. This can be illustrated by the examples in Figure 4.2. The first parameter of the recursive definition *goal* has a cyclic parameter link in both examples. A list can pass through the **cyclic parameter** link in Program (a) while a structured data can pass through the **cyclic parameter link**

Program (a) data link formed by "list"	Program (b) data link formed by "structured data"
goal([ ], stop). goal([X|L],N) :- goal(L,N).	goal(f( ), stop). goal(f(X),N) :- goal(X,N).

Figure 4.2

in Program (b). A data link is established in both cases. However, these data links are very different from the one built from subgoals.

(1) For the data link established through subgoals, there must exist at least a subgoal, besides the recursive subgoal, in the cyclic parameter link which is defined by an appropriate procedure; on the other hand, the data link established through special parameters may not have any subgoal in the cyclic parameter link at all.

(2) The implication of (1) is: When a data link is formed purely by subgoals, one can always find the values passed in and out the data link by merely analyzing the recursive definition. But there is no simple way to predict what data can pass through the data link established through special parameters.

One can simply analyze the recursive definition in Program (a) of Figure 4.1 to find out the values passed in and out of the data link. Because the only cyclic parameter link in the recursive definition exists between the first parameter in the head *X* and the first parameter in the recursive subgoal *P*, data links can only be established between parameter *X* and parameter *P*. As shown above, data can pass from *X* to *P* only if the subgoals *link1(X,A)* and *link2(A,P)* are instantiated as *link1(2,3)* and *link2(3,4)* respectively. Thus the possible values that can pass in and out of the data link are *2* and *4*. By merely analyzing the recursive definition, we can discover the particular values passing through the cyclic parameter link in a particular data link. On the other hand, when the recursive definition in Program(a) of Figure 4.2 is analyzed, only two points can be sure: (1) all lists except the empty list can pass through, and (2) the list passed to the next level of recursion will be one element less than the list in the present level. Analysis of Program (b) in Figure 4.2 gives a similar result: (1) all structured data with functor *f* can pass through except *f()*, and (2) the structured data passed to the next level of recursion will be one functor less than the one in the present level, eg., if the structured data is *f(f(f(1)))* in the present level of recursion, the structured data in the next level will be *f(f(1))*. Therefore, the data links in the recursive definitions established through special parameters have different properties from the data links in the recursive definitions established through subgoals. As shown by the examples in Figure 4.2, instead of particular values, only a set of possible values can be determined by analyzing the recursive definitions. Since the presence of this special data link will greatly complicate or even confuse our discussion, the case of the special parameters data links

will be examined after we explain how data links can indicate nontermination errors in a Prolog program. In the following sections, we shall limit our scope to data links formed by subgoals only. Then a general discussion will be given in Section 4.5.

To simplify our discussion, we shall divide the analysis into two steps corresponding to two types of recursive definitions. We shall first examine how to construct data links for subgoal cyclic parameter links in direct recursive definitions and then data links in indirect recursive definitions.

4.1.1 The Direct Recursive Definition Case

The case of direct recursive definition can be further broken down into two different sub-cases. The direct recursive definition with procedures consisting of facts alone requires a much simpler technique than the one with subgoal procedures consisting of some rules. We shall first show how to analyze the simpler case.

4.1.1.1 Subgoal Procedures with Facts Alone

If the subgoal procedures defining all the subgoals involved in a direct recursive definition consist of only facts, **data links will exist when some appropriate values are used as the particular arguments of the particular facts in these procedures.** In Figure 4.3, a cyclic parameter link is found between the parameter *X* and the para-meter *P* in the recursive definition. As shown in Figure 4.3, the subgoals that establish the cyclic **parameter link are** *link1*, *link3* and *link4*. To have a data link formed among these subgoals, **data must be**

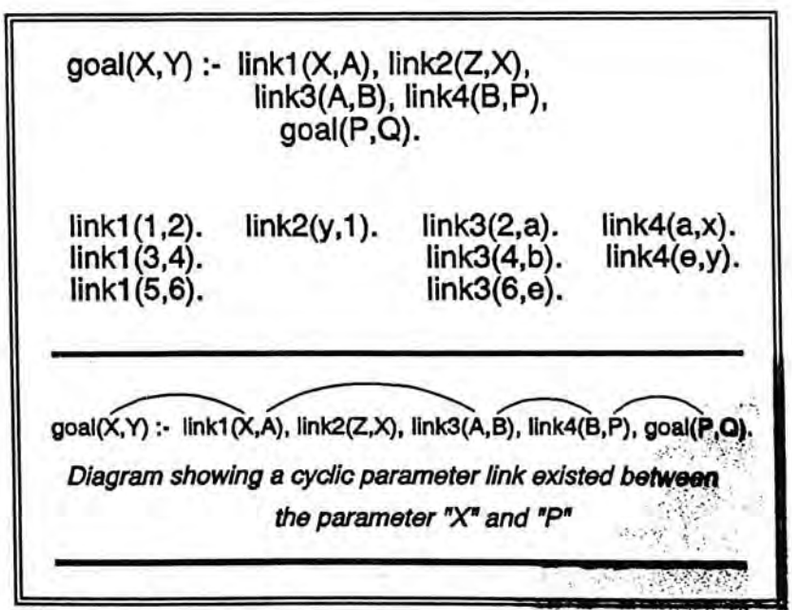


Figure 4.3

passed through these subgoals. Within the same rule, as shown in Chapter 3, data can only be transferred between the different subgoals through the parameters with the same name. Therefore, as indicated by those arcs in the diagram, data can pass from the subgoal *link1* to *link3* through the common parameter *A* and then from the subgoal *link3* to *link4* through the parameter *B*. However, it requires that the common parameter must be instantiated to the same value for all the subgoals sharing this parameter within this rule during its evaluation. If all the subgoals are defined by the procedures with facts alone, a data link can only be formed when some facts in these procedures have the same parameter values at the corresponding positions of the common parameter names in the subgoals of the recursive definition. Therefore, whether a data link can be formed depends on how the subgoal procedures have been defined by the facts.

For example, the second argument of some of those facts defining *link1* must be equal to the first argument of some of those facts defining *link3*. In Figure 4.3, the facts defining *link1* are *link1(1,2)*, *link1(3,4)* and *link1(5,6)* while those defining *link3* are *link3(2,a)*, *link3(4,b)* and *link3(6,e)*. Since the values *2*, *4* and *6* can be passed to the common parameter shared by the two subgoals, we can consider that **a partial data link** is formed between the subgoals *link1* and *link3*. Similarly, we can also conclude that a partial data link exists between *link3* and *link4* for the same reason. Then the next step is to show that a data link can be formed out of these partial data links; i.e., to show that these partial data links can be linked up.

To show how two partial data links can be linked up, we first consider a counter example. No data link can be formed if the definitions of *link1*, *link3* and *link4* in Figure 4.3 are changed to the following:

```
link1(1,2).        link3(2,a).        link4(b,5).
                   link3(a,b).        link4(b,7).
```

In this case, the fact *link1(1,2)* and the fact *link3(2,a)* can form a partial data link and this is also true for the fact *link3(a,b)* and *link4(b,7)*. Although there are two different facts in the procedure *link3* that can separately satisfy the partial data links *link1--link3* and *link3--link4*, none of them can satisfy the requirement to form a true data link

involving *link1*, *link3* and *link4*. Therefore, it is possible that the subgoal procedures can form some partial data links but not true data link. On the other hand, in the recursive definition in Figure 4.3, a data link can be formed between the parameters X and P because of the fact *link3(2,a)*. The first argument value *2* of *link3(2,a)* allow both subgoals *link1* and *link3* to have a common value for the shared parameter A while its second argument value *a* allow both subgoal *link3* and *link4* to have a common value for their common parameter B. As a result, a data link can be formed with the facts *link1(1,2)*, *link3(2,a)* and *link4(a,x)*. In this case, the fact *link3(2,a)* can be considered as a connector which can connect two partial data links. In conclusion, **two partial data links can be linked up if the common subgoal involved in both links can be bound to a connector.**

Apart from the procedures defining the subgoals in a cyclic parameter link, **the procedures defining the subgoals not included in the cyclic parameter link may become significant** in determining the existence of a data link. We can illustrate it by modifying the *link2* procedure. The procedures for subgoals *link1*, *link2*, *link3* and *link4* now becomes:

```
link1(1,2).      link2(y,2).      link3(2,a).      link4(a,x).
link1(3,4).                       link3(4,b).      link4(e,y).
link1(5,6).                       link3(6,e).
```

This modification can completely eliminate the data link originally existing in Figure 4.3. If we temporarily ignore the subgoal *link2*, we can construct four partial data links which can be linked up to form two data links. However, with the updated *link2*, no data link can in fact be formed because no data can be transferred in the recursive definition.

Due to the presence of the subgoal *link2*, which shares one of the parameters involved in the cyclic parameter link (i.e., X) so that the definition of *link2* can also determine what data can be transferred to the parameter X in the recursive rule. If the subgoals *link1*, *link3* and *link4* are instantiated with the facts *link1(1,2)*, *link3(2,a)* and *link4(a,x)* respectively, the second parameter X in the subgoal be instantiated with *1*. But the procedure of *link2* consists of the fact *link2(y,2)* alone. It causes the parameter

X to have value *2* only. Therefore, the subgoal *link1* can no longer be instantiated to *link1(1,2)*. So, there is no data link among the subgoals *link1*, *link3* and *link4*. Moreover, if the three subgoals are instantiated with the facts *link1(5,6)*, *link3(6,e)* and *link4(e,y)*, the parameter *X* now needs to be *5*. But this is also inconsistent with the procedure defining the subgoal *link2*. On the other hand, if the value *2* is supplied to the parameter *X* to be consistent with the procedure of *link2*, it then becomes inconsistent with the procedures defining the subgoals *link1*, *link3* and *link4*. Whatever value is instantiated to the parameter *X*, the evaluation of this recursive definition will always fail. So, no data can be transferred through the cyclic parameter link. Through this example, we can see that **the construction of a data link involves all the subgoals sharing those parameters included in the cyclic parameter link** rather than just the subgoals involved directly in the cyclic parameter link.

Therefore, detecting a data link in a recursive definition involves the following steps:

(1) Identify the cyclic parameter link and all those subgoals which are not part of the cyclic parameter link but share some common parameters with the subgoals in the cyclic parameter link.

(2) Construct partial data links between two subgoals with some **common parameters**.

(3) Examine all the partial data links sharing a certain **common subgoal** and see whether each common subgoal in two partial data links can be connected to a connector (i.e., a **common fact**).

(4) Examine those subgoals that are not involved in the cyclic parameter link but share some common parameters with some subgoals in the cyclic parameter link and check whether the common parameters can all be bound to some **common values**.

4.1.1.2 Procedures with Rules

So far we have discussed the data link formed by the subgoals **defined by facts** only. But subgoals can also be defined by rules alone or both rules and **facts. In the**

latter case, a data link can also be established in a similar but more complicated way. For a procedure consisting of some rules, the subgoals of these rules must be defined by some other procedures. Apart from the procedures defining the subgoals which form the cyclic parameter link, we must also examine the procedure of each subgoal involved in the rules. This process is repeated until a level is reached where all subgoals used in a rule are defined completely by facts.

In Figure 4.4, a cyclic parameter link is formed between the parameter *Y* and *Q* through the subgoals *assemble1*, *assemble2* and *finish*. To verify whether a data link exists in this cyclic parameter link, we must examine each subgoal involved in the cyclic parameter link. In order to find whether a partial data link can be formed between the subgoals *assemble2* and *finish*, we can directly analyze the procedures defining them as discussed in Section 4.2.1.1. as the subgoals are defined by facts alone. We can then easily notice that two partial data links exist. They are established through the facts *assemble2(part_a, part_b), finish(part_b, part_1)* and the facts *assemble2(part_c, part_f), finish(part_f, part_a)*. However, the existence of a partial data link between the subgoals *assemble1* and *assemble2* cannot be determined in this way. There is a rule in the procedure defining the subgoal *assemble1*. Although it is obvious that the fact *assemble1(part_a, part_4)* does not supply appropriate data to form a partial data link between *assemble1* and *assemble2*, we need a method to analyze the rule to see whether the rule can supply data to form a partial data link.

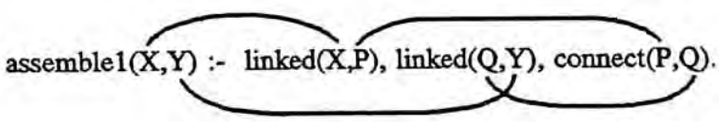
```
build(stop,part_end).
build(X,Y) :- assemble1(Y,A), assemble2(A,B), finish(B,Q),
              build(P,Q).

assemble1(part_a, part_4).
assemble1(X,Y) :- linked(X,P), linked(Q,Y), connect(P,Q).

assemble2(part_a, part_b).          finish(part_b, part_1).
assemble2(part_b, part_d).          finish(part_f, part_a).
assemble2(part_c, part_f).

linked(part_c, part_1).     linked(part_2, part_a).
linked(part_d, part_1).     linked(part_4, part_b).
linked(part_e, part_2).     linked(part_6, part_c).

connect(part_1, part_2).
connect(part_4, part_6).
```

Figure 4.4

In order to determine whether a partial data link can be formed between the subgoal *assemble1(Y,A)* and the subgoal *assemble2(A,B)*, we must know whether data can be transferred through the subgoal parameter link formed between these two

subgoals in the rule *build*. By analyzing the procedure defining *assemble2*, we already know which values can be assigned to parameters A and B in the subgoal *assemble2*. But we must also determine which values can be assigned to parameters Y and A in the subgoal *assemble1*. This is tantamount to finding out what values can be assigned to the parameters X and Y in the head of the rule defining the subgoal *assemble1*.

In our data transfer analogy, to assign a value to a parameter of a subgoal in a rule can be viewed as to transfer data through this parameter to the subgoals in the rule. This is similar to the steps followed in detecting data links mentioned in the previous section. Actually, the steps for subgoals defined by rules are the same except for the first one. For the previous case, in the first step to detect data links, we need to identify the cyclic parameter link. For the present case, the first step requires us to identify a **potential transfer link**. To transfer data from one subgoal to another in the same rule, there must be some common parameters. A potential transfer link indicates all the subgoals that are linked up by some common parameters to allow data to be transferred from one parameter to another in the same rule. Therefore, in the example in Figure 4.4, there is a potential transfer link between the parameter X and the parameter Y as shown in the following diagram:



The arcs show how a potential transfer link is formed among those subgoals through some common parameters. The potential transfer link in the rule *assemble1* involves all its three subgoals.

The next step is to construct **partial transfer links**, which are similar to partial data links. While a partial data link indicates that two subgoals in a cyclic parameter link are joined by a common parameter, a partial transfer link is formed between two subgoals with a common parameter in a potential transfer link. There are two partial transfer links in our example: between the first *linked* and *connect* and between the second *linked* and *connect*. The former partial transfer link is formed by the facts as:

linked(part_c, part_1), and connect(part_1, part_2).
linked(part_d, part_1),

and the latter partial transfer link is formed by the facts as:

linked(part_2,part_a) and connect(part_1,part_2).
linked(part_6,part_c). connect(part_4,part_6).

Examining these partial transfer links, we can now decide whether a **transfer link** exists. A transfer link exists among all the partial transfer links if there are common subgoals to link up the partial transfer links and these common subgoals can be bound to at least one fact. In this example, the common subgoal is the subgoal *connect* and there is a fact in the procedure defining *connect* to link these two partial transfer links together. It is the fact *connect(part_1, part_2)*. Therefore, a transfer link exists among the first and second *linked* and *connect* so that a transfer link also exists between the two parameters *X* and *Y*. The last step is to determine what data can be transferred through the transfer link between parameters *X* and *Y*. By examining all the facts used in forming the partial transfer links, we find that only the facts below are consistent in forming the transfer link among all three subgoals in the rule assemble1:

linked(part_c,part_1). linked(part_d,part_1).
connect(part_1,part_2). linked(part_2,part_a).

Therefore, we can conclude that the values that can be assigned to the parameter *X* are: *part_c* and *part_d* while the value that can be assigned to the parameter Y *is:* part_a. The process of determining the values that can be assigned to a subgoal defined by a rule can be summarized as follows:

(1) Identify the **potential transfer link** and all those subgoals which are not part of it but share some common parameters with the subgoals involved in the potential transfer link.

(2) Construct **partial transfer links** between two subgoals with some **common parameters.**

(3) Examine all the partial transfer links sharing a certain **common subgoal** and check whether each common subgoal can be bound to the **same fact.**

(4) Examine those subgoals that are not involved in the transfer link but share some common parameter with some subgoals in the transfer link and check whether the common parameters can be instantiated to a set of **common values**. If the requirements in both steps (3) and (4) are fulfilled, a **transfer link** exists and the values assigned to a particular parameter are equal to the values used in the facts that make the transfer link possible.

Moreover, the more complicated situation can be handled in a similar manner. If the procedure defining any subgoal of any rule used in a definition also involves some rules, we just need to take care of the more fundamental rules first. For example, if the recursive definition in Figure 4.4 remains the same except that the definition of the subgoal *assemble1* is changed to:

```
assemble1(part_a, part_4).
assemble1(X, Y) :- link(X, Y, P, Q), connect(P,Q).
link(X, Y, P, Q) :- linked(X,P), linked(Q,Y).
```

We can see that we need to first deal with the rule *link(X,Y,P,Q) :- linked(X,P), linked(Q,Y)*. However, the above process can again be applied to this rule without the need for any modification. The same result can be obtained as in the original example of the recursive definition in Figure 4.4. This process can work in a recursive manner. It recurs at the point where the subgoal is defined with some rules until a subgoal defined by all facts is reached. Then the process is applied to this rule (with subgoals defined by all facts) to find out all the possible values that can be transferred through the subgoal defining by this rule. If this subgoal is also a part of a rule that defines another subgoal, the process continues until it reaches the rule that forms the recursive definition. In our example, the process stops when it solves the subgoal *assemble1* in the rule build which forms the recursive definition.

4.1.2 The Indirect Recursive Definition Case

In an indirect recursive definition, a cyclic parameter link extends over more than one rule. To achieve requirement (2) of the data link in such a situation, it requires the condition that data can pass from the data link in one rule to the data link in the other rule. As described in Chapter 3, data can be transferred between two rules through the parameters at the same position in the parameter list of the recursive subgoal and the head of the next rule. Thus a data link can be established if each rule involved in the indirect recursive definition can have a data link and the data link of each rule can be linked up by the parameter that passes data from one rule to another. In other words, we can **consider each rule in the indirect recursive definition separately** and then examine whether data links found in each rule can be joined together to form a data link of the entire recursive definition.

In Figure 4.5, the indirect recursive definition has a cyclic parameter link shown by the arcs in the diagram. Actually, the indirect recursive definition in Figure 4.5 is equivalent to the direct recursive definition in Figure 4.3. In the rule *indirect_recur*, there is only one (rather than three) data link formed by the facts *link_level1(1, 2)* between the subgoals *link_level1* and *next_level* due to the limitation imposed by the subgoal *not_link*. For the rule *next_level*, two data links can be formed by the facts *link_ level2a(2,a), link_level2b(a, x)* and *link_ level2a(6, e), link_level2b(e, y)*. However, if the entire recursive definition is

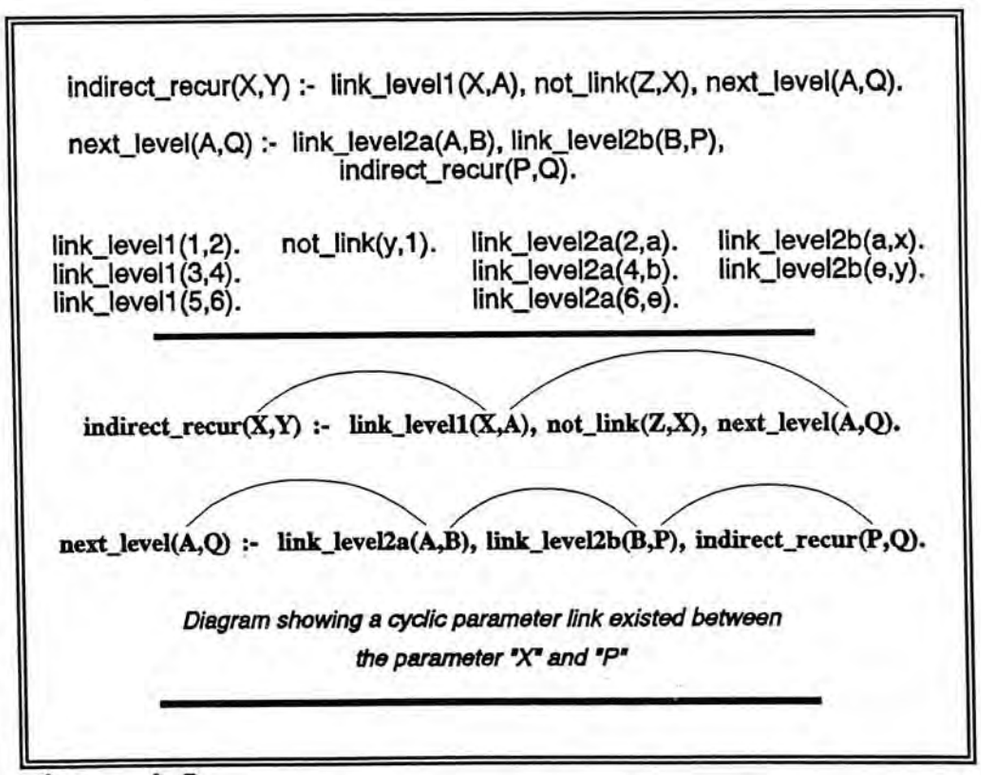


Figure 4.5

considered, there is only one data link that consists of facts: *link_ level1(1, 2), link_level-2a(2,a), link_level2b(a,x).*

By analyzing the data links in these two rules, we can see why only one data link can be formed. According to the procedure defining *link_level1*, we can find that the data link in the rule *indirect_recur* can pass only the value *2* to the data link in the rule *next_level*. However, the procedure defining *link_level2a* can allow the data link in the rule *next_level* to accept only values *2* or *6*. (The values are not *2*, *4* and *6* since the concern is not the subgoal *link_level2a* but the data link formed in the whole rule.) So only the value *2* can be successfully transferred from the data link in the rule *indirect_recur* to the data link in the rule *next_level*. Hence, only one data link can be formed in this indirect recursive definition.

This example shows how data links can be constructed from the different rules involved in an indirect recursive definition: first, we try to construct data links in different rules separately; then we analyze the values passing through the data links of each rule to see **whether some common values exist in the different data links of different rules**. In this example, the common value that allows a data link to form is the value *2* for the parameter *A* in the rule *next_level* and the rule *indirect_recur*. Therefore, there will be no data link in this indirect recursive definition if we make a little change in the procedures of subgoals used in the second rule as follows:

```
link_level2a(2,b).          link_level2b(a,x).
link_level2a(4,a).          link_level2b(e,y).
link_level2a(6,e).
```

Although two data links still exist in the rule *next_level*, the values passing through the data links are *4*, *x* and *6*, *y* for the parameters *A,P*. Thus no common value can be used to connect the data link in the first rule and the data link in the second rule.

In this example, except for the recursive subgoal, all other subgoals are defined by facts alone. The example is so constructed to keep the illustration as clear as possible. However, the method outlined in this section can be combined with the

transfer link constructing method described in the above section to handle those indirect recursive definitions with subgoals defined by rules only (or by both rules and facts). In the first step of detecting a data link in an indirect recursive definition, we consider each rule defining this indirect recursive definition separately. If any subgoal in one of these rules (that are defined by some rules) is encountered, we can examine the transfer link (if there is any) to determine the values that can be assigned to the parameters of this subgoal. With these values, we can move on to analyze the cyclic parameter link to detect the data link of a particular rule as what is done in the case of indirect recursive definitions with subgoals of facts alone. Then we examine all the data links in every rule used in the indirect recursive definition to determine whether there exists a data link in the indirect recursive definition. By applying the strategies in Section 4.2.1.1 and Section 4.2.1.2, we can detect the presence of any data link in each rule used in an indirect recursive definition and also the values passing through the data link (if any) no matter the subgoals involved are defined by facts or rules.

4.2 on the Difference between Pure and General Prolog

The above discussion on how a data link can be established shows why it is possible to detect data links in pure Prolog programs by analyzing the procedures of those subgoals involved in the recursive rule without the need for semantic knowledge. But the situation will be different if general Prolog programs are considered. **The difference between pure Prolog and general Prolog arises from the presence of *built-in predicates* in general Prolog.**

The procedure defining the *built-in predicate* is not provided by the programmer. Since the *built-in predicates* are not defined by any procedure in the program, the values that can be unified with the arguments of these *built-in predicates* are not specified in the program. Instead, the data passing through a cyclic parameter link formed with the *built-in predicate* are calculated only when the recursive definition is evaluated. In other words, there is simply no procedure of *built-in predicate* for us to analyze. If we want to

find the data links in a certain recursive definition in general Prolog, some kinds of run-time tracing technique must be used to identify the kinds of data that can pass through a *built-in predicate*. Furthermore, semantic knowledge must be provided to guide the tracing because most *built-in predicates* have arguments that can be unified with an infinite set of values.

Figure 4.6 shows an example of how semantic knowledge is needed to find out a data link in a general Prolog recursive definition. There are two almost equivalent recursive definitions. Both have two cyclic parameter links between parameters *N1* and *N1_* and between parameters *N2* and *N2_*. Their semantics are almost identical. But only Program (a) has a much greater calculating power: Program (a) can theoretically add up any two numbers while Program (b) can only handle numbers from 1 to 10. The powerful calculating ability in Program (a) comes from the *built-in predicates*: *N1_ is N1 - 1* and *N2_ is N2 + 1*. Basically, *successor(N1_, N1)* and *successor(N2, N2_)* are semantically equivalent to *N1_ is N1 - 1* and *N2_ is N2 + 1* except for their limited range of inputs. **In the pure Prolog version, the subgoal *successor* is defined by a finite number of facts. This results in a limit on the range of values** that can be unified with variables *N1* and *N2*. This provides us a way to determine the existence of any data link in the recursive definition by analyzing the procedure that defines the subgoal *successor*.

Program(a) with "built-in" predicate

```
add2(0, N, N).
add2(N1, N2, R) :-
   N1_ is N1 - 1,
     N2_ is N2 + 1,
       add2(N1_, N2_, R).
```

Program(b) pure Prolog version

```
add2(0, N, N).
add2(N1, N2, R) :-
   successor(N1_,N1),
      successor(N2,N2_),
        add2(Nl_, N2_, R).

succesor(0,1).    successor(5,6).
succesor(1,2).    successor(6,7).
succesor(2,3).    successor(7,8).
succesor(3,4).    successor(8,9).
succesor(4,5).    successor(9,10).
```

Figure 4.6

However, there is no procedure for us to analyze when we attempt to find the data link in Program (a). We, therefore, can only conclude the existence of a data link between the parameter *N1* to *N1_* and *N2* to *N2_* by either one of two ways: (1) having semantic knowledge of these built-in predicates or (2) tracing the evaluation of the recursive definition. Once we understand that the predicate *Value is Expression*

calculates the *Expression* on its right hand side and then assigns the result to the variable on its left, we can show that some data can be transferred from the parameter *N1* to the parameter *N1_* by the predicate *N1_ is N1 - 1* during the recursion. Since a cyclic parameter link is also formed between parameters *N1* and *N1_*, a data link exists between them. **The semantic knowledge of this predicate can be used to determine *partially* whether a data link exists.** It is *partially* determined because, sometimes, it depends on the input value to determine any data transferring through the predicate. For example, if the *Expression* supplied to the predicate is *10 / N*, an error occurs when *N* is *0* and no data can be transferred through this predicate. On the other hand, we can also know whether any data can be transferred through a cyclic parameter link by simply tracing it during the evaluation. **But semantic knowledge may be required in some situations in the tracing.** In this case, the parameter *N1* must be first unified with some values before the predicate *N1_ is N1 - 1* is evaluated. The evaluation of subtracting 1 from an uninstantiated variable results in failure. Tracing without initializing the parameters *N1* and *N2* will not give a correct conclusion. Moreover, semantic knowledge may be required for choosing the initial values for the parameters involved in the tracing. In Program (a), it does not matter what values are chosen to initialize the parameters *N1* and *N2*. However, it becomes important if the recursive rule is modified as follows:

```
add2(N1, N2, R) :-  N1_ is N1 - 1, N2_ is N2 + 1,
                    N1_ > N2_, add2(N1_, N2_, R).
```

If the initial values of *N1* and *N2* are *1* and *2*, a simple tracing will detect that there is no data link in this recursive definition. But there will always exist a data link when *N1* is assigned a value greater than the value instantiated to *N2* by 3 or more. So, tracing requires semantic knowledge in this case.

In conclusion, although parameter analysis can be applied in the same way to both pure Prolog and general Prolog, data analysis cannot be applied to general Prolog in the same way as pure Prolog. To detect a data link in a general Prolog recursive definition, semantic knowledge must be provided. Data analysis, in general Prolog, needs to be conducted with an interactive or a run-time tracing technique. However, all

these problems in the case of general Prolog do not exist in the case of pure Prolog. Due to the absence of the built-in predicate in pure Prolog, one can detect nontermination with an analytical approach.

4.3 Data Link Significance

After we have discussed what a data link is and how it can be established in different situations, we move on to discuss the significance of data link. From the above discussions, we can see that a data link exists if some data can pass through a certain cyclic parameter link in one complete cycle. In other words, the presence of a data link guarantees that some values can pass through one parameter cycle of recursion (however, as explained in Section 4.1, one or more levels of recursion may be involved depending on the number of recursion levels involved in the corresponding cyclic parameter link).

Therefore, the significance of the data link is: **the existence of a data link indicates that the corresponding cyclic parameter link can act as a parameter modifying process in at least one parameter cycle of recursion.** On the other hand, the cyclic parameter link that cannot form a data link implies that it is not a parameter modifying process at all. The reason is simple: no parameter can be really modified if no value can be successfully transferred through a cyclic parameter link.

As has been pointed out in Chapter 3, to verify whether a cyclic parameter link is an exit-reaching process involves two steps: first, to show that some data can pass through this cyclic parameter link; second, to show that an exit condition exists in the cyclic parameter link. Although the presence of a data link cannot provide the necessary semantic knowledge to verify whether an exit condition is present in the corresponding cyclic parameter link, the absence of data link in a recursive definition indicates that the corresponding cyclic parameter link cannot include any *real* exit condition. Even though a cyclic parameter link may include an *intended* exit condition,

the absence of data link shows that no data can ever be modified through this cyclic parameter link. If no data can pass through a cyclic parameter link, the cyclic parameter link cannot act as an exit-reaching process and consequently the *intended* exit condition can never act as a *real* exit condition. In other words, **the absence of data link in a cyclic parameter link indicates that the cyclic parameter link is not an exit-reaching process.**

But there is one interesting point: **the absence of any data link does not imply the presence of nontermination error.** In contrast, the evaluation of such a recursive definition can always terminate. The absence of data links implies that no data can pass through some of the subgoals involved in the cyclic parameter link. Therefore, these subgoals always fail to be instantiated with any value. Any attempt to evaluate such a recursive definition can only result in failure. In fact, not even one level of recursion can proceed in the recursive definition that has no data link. Therefore, the apparent dilemma can be solved if we can distinguish the *exit condition in action* from the *semantic exit condition*. Since the absence of data links forms an *exit condition in action*, **it can terminate even though there is semantically no proper exit condition.**

This can be illustrated by the examples in Figure 4.7. In both Program (a) and Program (b) of Figure 4.7, the second rule of *go* forms a direct recursive definition with a cyclic parameter link between the first parameter of the rule head, X and the first parameter of the recursive subgoal, A. Although there is no real difference between these two programs in terms of the results of

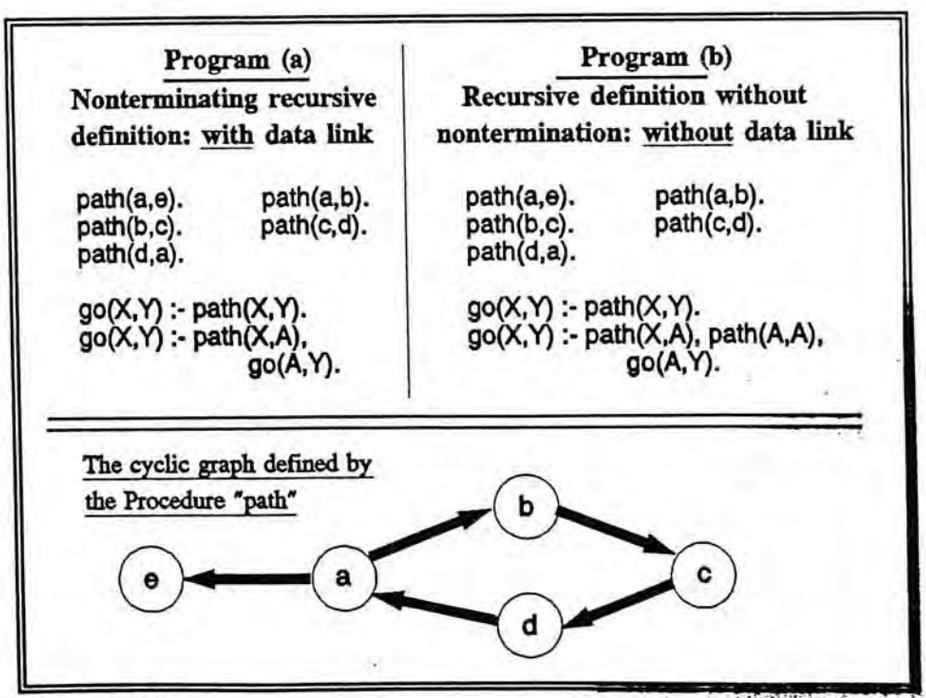


Figure 4.7

parameter analysis, they are greatly different from each other in terms of the results of data analysis. In Program (a), data can be transferred through the subgoal *path(X,A)* so that data links can be established between parameters *X* and *A*. However, in Program (b), the cyclic parameter link is formed by the two subgoals *path(X,A)* and *path(A,A)* instead of the subgoal *path(X,A)* in the case of Program (a). By analyzing the procedure defining *path*, it is clear that no fact in the procedure of the subgoal *path* can be instantiated with the subgoal *path(A, A)*. Therefore, no data link can be formed in Program (b) despite the great similarity between the two programs. In spite of the absence of data link, the evaluation of Program (b) will definitely come to an end. In fact, no evaluation can be carried out at all because any value supplied to this recursive definition will end up in failure at the subgoal *path(A,A)*. The problem is, not one of nontermination, but that the supposedly *recursive* definition *go* can never recur. In other words, there is a semantic error in this recursive definition.

On the other hand, while a data link shows that a cyclic parameter link can actually transfer data and modify them for at least one level of recursion, this is not the proof of termination. This is shown by Program (a) in Figure 4.7. Because the procedure of the subgoal *go* actually defines a cyclic graph as shown in the diagram inserted in Figure 4.7, the evaluation of Program (a) eventually leads to nontermination. Contrasting these two examples, we can see that a data link indicates whether a cyclic parameter link can allow any value to pass through itself in at least one level of recursion; its absence indicates that the corresponding recursive definition is not properly defined. In other words, nontermination cannot be detected by constructing data links alone; our data analysis is not complete until we take into consideration the cyclically and non-cyclically (or simply **cyclic and non-cyclic) connected data-link lists.** In the following sections, we shall discuss how nontermination in pure Prolog can be detected by examining whether data links found in a recursive definition can also form some connected data-link lists and what kind of connected data-link lists can be formed by these data links.

4.4 Connected Data-link Lists

4.4.1 Data Links and Connected Data-link Lists

A data link represents a data transfer in one parameter cycle of recursion through a cyclic parameter link. But in the nonterminating evaluation of a recursive definition, the data transfers in a cyclic parameter link during the recursion actually link up to form a infinitely long *data transfer sequence*. The length of a data transfer sequence in a certain cyclic parameter link depends on how many levels of recursion involved in one parameter cycle and how many parameter cycles can be completed during the evaluation. Therefore, a data transfer sequence can represent a data transfer through one or more parameter cycles of recursion. Since a data transfer sequence of one parameter cycle can be represented by a data link, several data links can be connected together to represent a data transfer sequence of multiple parameter cycles. The result of these connected data links is referred to as **a connected data-link list.**

In order to show how a connected data-link list can be formed from a set of data links, we can consider the situation of more than one data links for the same cyclic parameter link. The situation implies that there are several possible values that can be transferred through the same cyclic parameter link. Although each of these data links only indicates that certain values can be transferred through the recursive definition in only one parameter cycle of recursion, some of these data links in the same cyclic parameter link may be combined together to represent a data transfer sequence through more than one parameter cycles of recursion. Therefore, the concept of connected data-link lists is an extension of the concept of data links to describe the data transfer over **all** recursion levels in one complete recursion.

Therefore, the difference between a data link and a connected data-link list is only a difference of length, i.e., a difference in the number of recursion levels over which a data transfer takes place. On the other hand, the difference between the absence and the presence of data link is great; it is tantamount to the difference between a proper recursive definition and an improper one. Hence, we must not ignore

the data transfer sequence of one single parameter cycle of recursion. In other words, a data link must be considered as a special case of connected data-link list. The following discussion on the cyclic and non-cyclic connected data-link list is therefore also applicable to data links.

The example in Figure 4.8 illustrates how a connected data-link list can be constructed out of some simple data links. In Figure 4.8, there is a simple recursive definition with a cyclic parameter link between parameters *X* and *P* with only the subgoal *link* involved in the cyclic parameter link. According to the procedure defining the subgoal *link*, there are six data links between *X* and *P*. They are formed by the facts *link(1,2)*, *link(3, 4)*, *link(5, 6)*, *link(2,a)*, *link(4, b)* and *link(b,5)*. In considering the data link from the viewpoint of the data transfer analogy, we can consider that the values *1*, *3*, *5*, *2*, *4* and *b* are passed into the cyclic parameter link through the parameter *X* with values *2*, *4*, *6*, *a* and *5* respectively coming out from the parameter *P*. Therefore, we denote these data links in this example as *1--2*, *3--4*, *5--6*, *2--a*, *4--b* and *b--5*. If we just link up any two data links, with the tail of one data link and the head of the other one having the same value as *1--2* and *2--a* respectively, we can form a connected data-link list as *1--2--a*. Six connected data-link lists can be formed in the example. They are *1--2--a*, *3--4--b--5--6*, *5--6*, *2--a*, *4--b--5--6* and *b--5--6*.

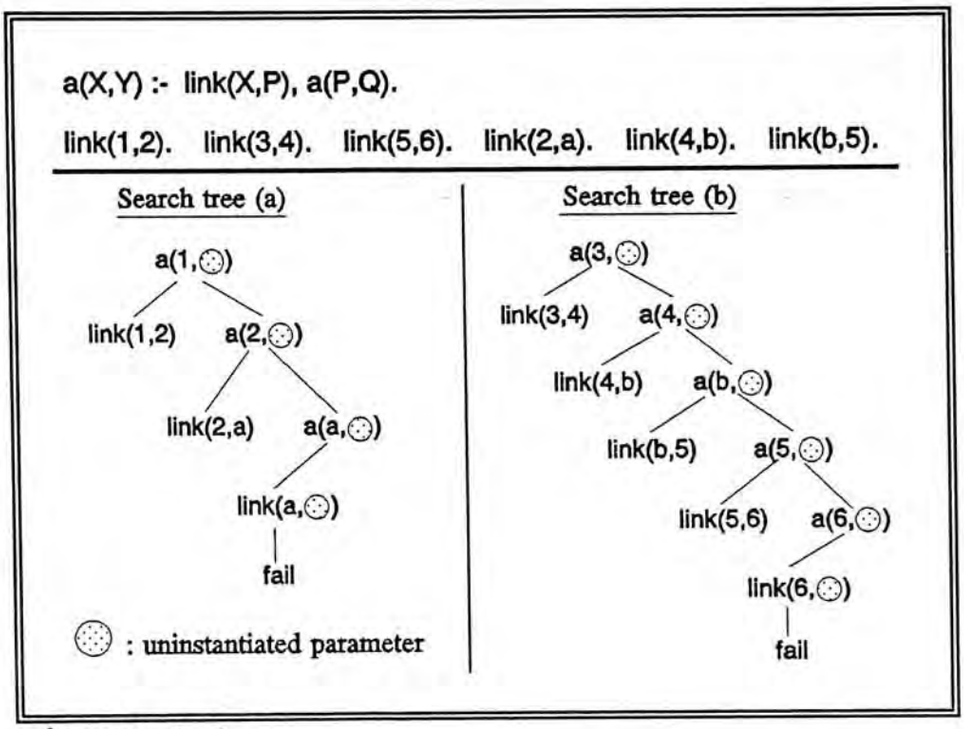


Figure 4.8

If we examine how a recursive definition is evaluated, we can find a great similarity between connected data-link lists and the values passing through each level of

recursion. For example, on the one hand, if the query *?- a(1,X)* is supplied, a two-level recursion occurs with a calling sequence of the goal *a* as *a(1,X), a(2,X), a(a, X)*, (where *X* is the uninstantiated parameter) as shown by the search tree (a) in Figure 4.8. On the other hand, if we identify the connected data-link list which is started by *1*, we can find one connected data-link list with a length of two data links: *1--2--a*. Moreover, if the query *?-a(3,X)* is supplied, it results in a calling sequence of the goal *a* as *a(3,X), a(4,X), a(b,X) a(5,X), a(6,X)* (where *X* is the uninstantiated parameter) and then the goal fails. The number of recursion levels is four. We also have a connected data-link list with a length of four data links started by *3*: *3--4--b--5--6*. We can find **the one-to-one correspondence between a data transfer sequence and the connected data-link list.**

Although the data transfer sequence is different when different queries are supplied, our connected data-link list technique has no difficulty in predicting whether a certain query can have a data transfer sequence and what this connected data-link list looks like. For example, if the query *?-a(5,X)* is supplied, we can simply check the connected data-link lists started by *5*. There exists a connected data-link list *5--6*, which indicates that this query can have a data transfer sequence which is: a(5,X), a(6, X) and then it fails. If the query *?-a(7,X)* is supplied, our technique shows that no data transfer occurs because there is no connected data-link list started with *7*. Both predictions can be confirmed by drawing search trees for the two queries.

Before we continue our discussion on connected data-link lists, we must note that the data links and also the possible connected data-link lists for **a cyclic parameter can extend to more than one level of recursion.** In the above example in Figure 4.8, we only consider the data links in the single-level cyclic parameter link. In this case, the data link is always one level long. However, in the case of multi-level cyclic parameter link, any data link formed must also extend over several levels of recursion. In Program (b) in Figure 4.9, the cyclic parameter link is actually two levels long. The difference between a multi-level cyclic parameter link and a single-level cyclic parameter link is obvious if Program (a) and Program (b) are compared. If we are only concerned with the change of data when passing one cycle of the cyclic parameter link, the data links can be expressed as *1--3*, *3--5* and *5--1*. On the other hand, if the change of data in

data in every level of recursion is considered, it is better to express the data links as *1--2--3*, *3--4--5* and *5--6--1*. Both can work well in our discussion on how to construct connected data-link lists out of data links. However, because our discussion is based on the concept of cyclic parameter links and the concept of parameter cycles of recursion, we shall denote a data link with the change of data in a complete parameter cycle instead of with the change of data in each level of recursion. Therefore, both single-level cyclic parameter link in Program (a) and multi-level cyclic parameter link in Program (b) in Figure 4.9 have the same notation of their data links:*1--3*, *3--5* and *5--1*. Since there is no significant difference between a single-level data link and a multi-level data link, the following discussion on connected data-link lists will be based on a data link of one level in order to simplify our discussion.

Program (a)

```
goal(AX,BX) :- link(AX,XA), goal(XA,XB).

link(1,3).  link(3,5).  link(5,1).
```

Program (b)

```
goal(AX,BX) :- linka(AX,XB), linkb(BX,XA),
               goal(XA,XB).

linka(1,2).     linkb(2,3).
linka(3,4).     linkb(4,5).
linka(5,6).     linkb(6,1).
```

Figure 4.9

4.4.1.1 Connected Data-link Lists and Data Transfer Sequences

By comparing the sequence of values passing through the consecutive parameter cycles of recursion with the connected data-link list, we can see that **the connected data-link list represents exactly the same sequence of values passing through the cyclic parameter link during each successive parameter cycle of recursion.** The relation between them is obvious:

(1) A data link indicates that some data can pass through a certain cyclic parameter link in one parameter cycle of recursion.

(2) The first value shown in a data link indicates what value is passed into this parameter cycle of recursion.

(3) The second value in a data link shows what value can come out after this parameter cycle of recursion.

(4) The cyclic parameter link implies that the value coming out of the parameter cycle of recursion can be transferred to the same parameter in the next parameter cycle of recursion.

(5) **Therefore, two data links that can be linked up together indicate that some data can pass through two consecutive parameter cycles of recursion.**

This is the reason why we only link up two data links with a common value. The common value implies that this value passing out from one parameter cycle of recursion can be immediately fed to the same parameter in the next parameter cycle of recursion. For example, in Figure 4.8, there is a connected data-link list of *3--4--b--5--6*. Since four data links are involved and one parameter cycle of recursion of the data links only involves one level of recursion, we can predict that a four levels of recursion will result and the first parameter of the goal *a* will be instantiated with values of *3*, *4*, *b*, *5* and *6* during successive level of recursion. This is just confirmed by Search tree (b) in Figure 4.8.

In addition to the data transfer sequence, **the connected data-link list can also indicate the maximum number of levels of recursion that can be reached before termination occurs.** By examining the levels of recursion indicated in Search trees (a) and (b), we can also find that the maximum number of levels of recursion can be reflected by the length of the connected data-link list. The longest connected data-link list can indicate the maximum number of recursion levels that can be reached. The length of a connected data-link list indicates how many parameter cycles of recursion can be reached by the corresponding data transfer sequence. If the number of recursion levels involved in one parameter cycle is known, we can calculate the length of the connected data-link list in terms of levels of recursion. In Figure 4.8, the parameter cycle of each data link involves only one level of recursion. Therefore, the maximum number of recursion levels that can be reached by the connected data-link list that has a length of two data links is two levels of recursion while a connected data-link list of four data links long can represents a data transfer sequence of four levels of recursion. This is clearly shown by Search trees (a) and (b) in Figure 4.8. The reason for this

relationship between the length of connected data-link list and the number of levels of recursion is the same reason for the relationship between a connected data-link list and the value transfer sequence of the successive level of recursion.

As discussed in Sections 4.1 and 4.3, a data link indicates what kind of values can pass into and out of a parameter cycle of recursion. But data transfer in one parameter cycle of recursion, as shown in this chapter and the last chapter, is completely determined (if we put aside the case of special parameters for a while) by the definitions of the subgoals forming the cyclic parameter link. Therefore the connected data-link list in fact represents the data transfer sequence made possible by the definitions of subgoals involved in a certain cyclic parameter link. It explains why there is a correspondence between the data transfer sequence and the connected data-link list: (1) correspondence between the number of parameter cycles of recursion and the length of connected data-link list, (2) correspondence between the values in the data transfer sequence and the values in the connected data-link list, and (3) correspondence between the order of values in the data transfer sequence and the order of values in the connected da-link set.

4.4.1.2 Connected Data-link Lists and Backtracking

Moreover, a connected data-link list can accurately account for a data transfer sequence even if **backtracking** happens. In Prolog, the backtracking mechanism always causes all possible data transfer sequences to be tried. Backtracking occurs when a data transfer sequence comes to its end. Therefore the point where backtracking occurs is independent from other data transfer sequences. However, the point where evaluation is resumed, that is, the point where the backtracking mechanism leads the evaluation to, is affected by other data transfer sequences.

In Prolog, the backtracking mechanism always tries to resume evaluation at the latest level of recursion if it is possible. If it is not, it backtracks to the previous level. That is, if necessary, it can backtrack all levels, one by one, until level one is reached. Backtracking goes on until it can find another possible data transfer sequence or until

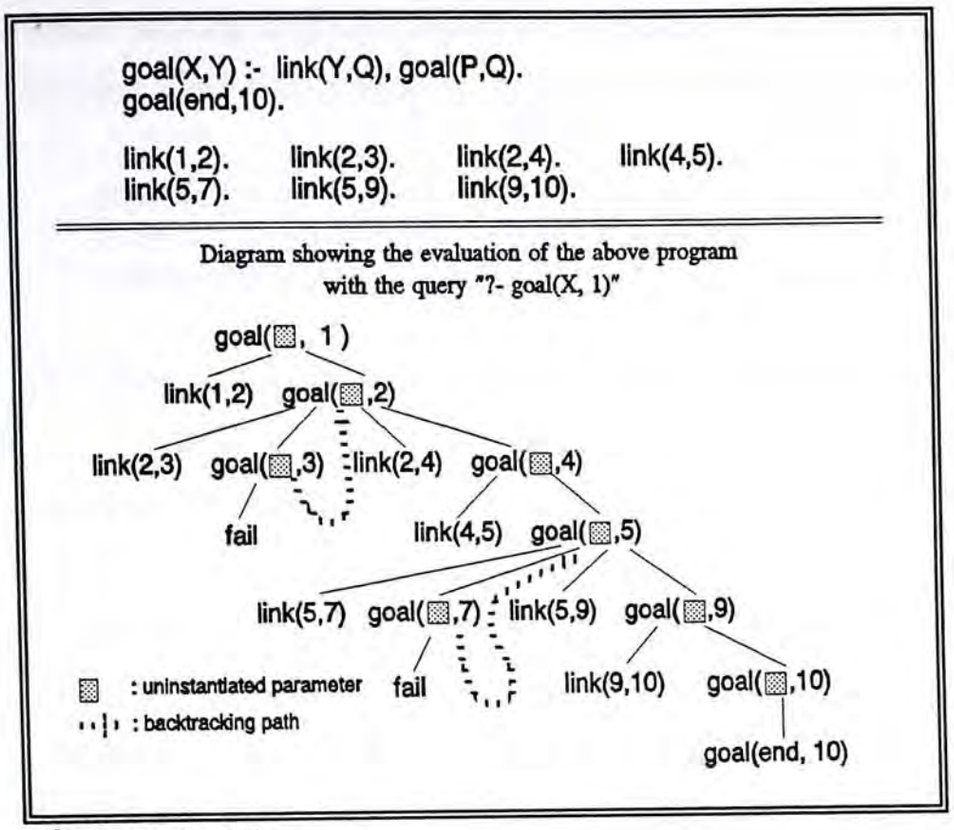


Figure 4.10

all possibilities are exhausted. Therefore, evaluation can be resumed at the very point where an alternate data transfer sequence is found. If an alternate data transfer sequence is found, it is not necessary to backtrack to the very beginning of the recursion. If a recursive definition has several data links all with the first value being the same, there are different possible values for the next level of recursion. It can be illustrated by the example in Figure 4.10. There are two data links, *2--3* and *2--4*, which are started by the same value *2*. If the value *2* is supplied to this level of recursion, the value transferred to the next level of recursion can be either *3* or *4*. Because of the backtracking mechanism in Prolog, both of the two possibilities will be tried eventually. It is shown by the search tree[1] in Figure 4.10. At the first level of recursion, after the data transfer sequence corresponding to *2--3* is tried, backtracking occurs and leads the evaluation to the point to try the data transfer sequence corresponding to *2--4*. Therefore we have some **data transfer sequences which have a common segment**. The point where backtracking resumes the evaluation of a recursive definition is the point

1 This search tree does not show further searches after the unification of "goal([uninstantiated parameter], 10)" with the fact "goal(end, 10)" at the fifth level of recursion because it has already served our purpose here to illustrate the relationship between a connected-data link and backtracking. However, the backtracking mechanism will continue to search for other possible solutions for the parameter "X" with the parameter "Y" instantiated with "1". But all the subsequent searches will result in failure.

where the common segment ends. Of course, we can eliminate the common segment by viewing these data transfer sequences as a single data transfer sequence as follows:

> goal(X,Y), goal(X, 1), goal(X, 2), goal(X, 3), **backtrack**, goal(X, 4), goal(X, 5), goal(X, 7), **backtrack**, goal(X, 9), goal(X, 10) and goal(end, 10) where *X* is the uninstantiated parameter

However, this view is less satisfactory because it cannot show important information such as the number of parameter cycles of recursion and the point where backtracking resumes the evaluation.

Therefore, if we follow the view that there are several different data transfer sequences with a common segment instead of one single data transfer sequence, each connected data-link list can account for a data transfer sequence just as the case of no backtracking. In Figure 4.10, there is a recursive definition which has only one cyclic parameter link between the second parameter in the recursive rule head and the second parameter in the recursive subgoal. The data links in the recursive definition are *1--2*, *2--3*, *2--4*, *4--5*, *5--7*, *5--9* and *9--10*. Therefore, eleven connected data-link lists can be formed, they are *1--2--3*, *1--2--4--5--7*, *1--2--4--5--9--10*, *2--3*, *2--4--5--7*, *2--4--5--9--10*, *4--5--7*, *4--5--9--10*, *5--7*, *5--9--10* and *9--10*. With the query *?-goal(X,1)*, there are three possible data transfer sequences since there are three connected data-link lists started by *1*: *1--2--3*, *1--2--4--5--7* and *1--2--4--5--9--10*. By comparing them with the search tree in Figure 4.10, we can see the common segment in the connected data-link lists correspond to the common segment in the data transfer sequences. And the length of and the values in the connected data-link list can also show the number of parameter cycles of recursion and the values being transferred in the corresponding data transfer sequence as usual.

4.4.1.3 Connected Data-link Lists and the Recursion Result

The example in Figure 4.10 also shows that a data transfer sequence in a recursive definition does not guarantee that the evaluation of this recursive definition can succeed. In other words, as data transfer sequences are represented by connected

data-link lists, **the presence of a connected data-link list in a recursive definition does not guarantee the evaluation of this recursive definition will succeed.** For example, the data transfer sequences corresponding to the connected data-link lists *1--2--3* and *:1--2--4--5--7* all lead the recursion to failure. The recursion can only succeed when the data transfer sequence can assign a value that can agree with some facts in the procedure that defines the recursive goal. In this example, the value *10* can agree with the fact *goal(end, 10)* so that the recursion succeeds. (However, the solution of the parameter *X* is still an uninstantiated parameter since there is no data link to transfer the value *end* back.)

Therefore, **a connected data-link list can also be used to check whether a recursive definition can succeed.** If a recursive definition has a connected data-link list which has a tail value that can agree with a certain value of one of the facts defining the recursive subgoal, the evaluation of this recursive definition can succeed. Therefore, if we change the fact in the procedure *goal* in Figure 4.10 from *goal(end,10)* to *goal(end,5)*, we can still conclude that the recursion in Figure 4.10 can succeed by analyzing the connected data-link lists. (There are three connected data-link lists that end in value *5*: *4--5*, *2--4--5* and *1--2--4--5*.)

4.4.2 Cyclic and Non-Cyclic Connected Data-link Lists

Besides the connected data-link list of finite length, the data links from some recursive definitions can form a connected data-link list of infinite length. We shall refer to the finite set as a **non-cyclic connected data-link list** while the infinite one is referred as **cyclic connected data-link list.** Actually, we have an example of cyclic connected data-link list back in Figure 4.7. A cyclic connected data-link list exists in Program (a). Comparing it with the example in Figure 4.8, we can find great similarity between them. Although the rule *go* in Program (a) of Figure 4.7 is defined as:

```
go(X,Y) :- path(X,A), go(A,Y).
```

and the rule *a* in Figure 4.8 is defined as:

a(X,Y) :- link(X,P), a(P,Q).

They are the same if we can put aside the naming of the parameters and the subgoals for a while. The rule *a* is exactly the same as the rule *go* if we replace a with *goal*, *link* with *path*, *P* with *A*, and *Q* with *Y*. In Prolog, naming does not play an important role in the evaluation of a program as long as the changes are consistent. Therefore, we can consider that both of them have the same recursive rule and the same cyclic parameter link. But there is a significant difference between them when we consider the procedures defining the subgoals *path* and *link* in these two examples. They result in two different sets of data links in these two different examples. The data links in Figure 4.8 are *1--2*, *3--4*, *5--6*, *2--a*, *4--b* and *b--5* while the data links in Program (a) of Figure 4.7 are *a--e*, *a--b*, *d--c*, *c--d*, and *d--a*. They are two distinctively different sets of data links. As shown above, the first set of data links can just form two chains with finite length. But if we try to link up the data links in Program (a) of Figure 4.7 as what has been discussed above, we shall soon find ourselves engaged **in an infinite task.** Suppose we start with the data link *a--e* ,we first only form a single data link chain. But if we start with the data link *a--b,* the connected data-link list can be infinite as *a--b--c--d--a--b--c--d--a--* ... . However, **it is obvious that this infinite sequence is constructed by repeating a segment of *a--b--c--d* and it can be represented as a cycle of *a--b--c--d--a.*** This is the reason why we refer to such a connected data-link list of infinite length as a cyclic connected data-link list. In contrast to the cyclic connected data-link list, we refer to the finite connected data-link list as a non-cyclic connected data-link list because we cannot find a repeating segment in it.

With the idea of cyclic and non-cyclic connected data-link lists, we can proceed to develop our technique for detecting nontermination without the need of semantic knowledge in pure Prolog. As shown in the above discussion, the need for semantic knowledge occurs only after a cyclic parameter link is found. In our approach developed so far, one can conclude from the absence of any cyclic parameter link that nontermination will occur. Further examination, however, is required to confirm whether nontermination will occur when some cyclic parameter links are detected. As shown in Chapter 3, a cyclic parameter link is a potential exit-reaching process. Hence, a definite conclusion can be drawn only after we can verify whether any of the cyclic

parameter links found can work as an exit-reaching process. At first glance, semantic knowledge seems to be needed to determine which is the exit condition in the recursive definition in order to decide whether a cyclic parameter link can act as an exit-reaching process. Moreover, it also seems necessary to have semantic knowledge of how the data passing through the exit condition is modified during the process since a cyclic parameter link can act as an exit-reaching process only if the exit condition can be met at a certain point during the evaluation of the recursion.

However, **the concepts of cyclic and non-cyclic connected data-link lists can be used to develop a new approach to detect nontermination in pure Prolog programs without the need of any semantic knowledge.** According to the cyclic or non-cyclic connected data-link list formed out of the data links in a pure Prolog program, we can bypass the verification steps that require the semantic knowledge of this program. In the following sections, we shall show that **nontermination will arise if all the cyclic parameter links in one recursive definition have at least one cyclic connected data-link list.** To understand why the cyclic or non-cyclic connected data-link list can be used to detect nontermination in Prolog, we shall examine the relationship among cyclic connected data-link list, non-cyclic connected data-link list and exit-reaching process.

4.4.2.1 Non-Cyclic Connected Data-link Lists and Exit Conditions

As discussed in Chapter 2, the **exit condition** of a recursive definition in Prolog is simply formed by **some subgoals or special parameters** (i.e, lists or structured data) **which can block the evaluation of a recursive definition at a certain point in its evaluation.** In Chapter 2, we have also shown how a subgoal or special parameter can work this way. At a certain point in the recursion, one or more subgoals[1] fail or some special parameter becomes non-unifiable in all situations. Therefore the recursive subgoal cannot be reached and the next level of recursion is stopped. Because of the

1. One such subgoal is a necessary and sufficient condition to block further recursion. If a programmer puts more than one of such subgoals in a recursive definition, only the one that is first encountered during the course of the recursion is actually effective.

reason given in Section 4.1, we limit our discussion at this point to only the case of subgoals. **On the other hand, if a recursive definition is properly defined, there must be one or more levels of recursion** before the evaluation of this recursive definition is blocked by such a subgoal. In other words, before this subgoal can act as an exit condition, some data can also pass through this subgoal in one or more levels of recursion. **Therefore, the role of an exit condition and the role of data transfer can be accomplished by the same subgoal without any conflict.** This point is important to an understanding of how an exit-reaching process can be a parameter modifying process at the same time.

Because this subgoal is part of the exit condition, it must also **be part of the exit-reaching process.** Since a cyclic parameter link is a potential exit-reaching process, this subgoal must **form part of the cyclic parameter link.** On the other hand, a proper recursive definition must allow some levels of recursion to occur before it stops, some data have to be transferred through the subgoal that can later act as an exit condition in several levels of recursion. Consequently, data links must be constructed out of the subgoals involved in this cyclic parameter link. As one or more levels of recursion is a basic requirement for a proper recursive definition, these data links can always form some connected data-link lists.

In other words, **the subgoal that acts as the exit condition must be involved in a part of a certain connected data-link list.** Obviously, the data links obtained from a recursive definition with an exit-reaching process must be capable of being linked together to form a non-cyclic connected data-link list. Because the subgoal that can act as an exit condition stops further data transfer through itself when the exit condition is met, there is a point where no more data link can be linked up to the connected data-link list that has already been constructed. The length of the connected data-link list formed out of a terminating recursive definition must be finite. In other words, a recursion will terminate only if a non-cyclic connected data-link list is found in a recursive definition. One can conclude that **the presence of an exit condition (and an exit-reaching process) necessitates the existence of a non-cyclic connected data-link list in a recursive definition.**

On the other hand, the existence of some non-cyclic connected data-link lists in a cyclic parameter can also indicate that there is an exit-reaching process (and an exit condition) in a recursive definition. If at least one cyclic parameter link can be established in a recursive definition and the procedure defining the subgoals in this cyclic parameter link can form one or more non-cyclic connected data-link lists, this cyclic parameter link will act as an exit-reaching process (even though the programmer may not intend this cyclic parameter link to be an exit-reaching process). That is because this cyclic parameter link is the only way in the recursive definition to allow data transfer and the non-cyclic connected data-link list indicates that data transfer through this cyclic parameter link will be terminated at a certain point of the recursion. This implies that there will be a blockage in the data transfer process during the recursion. This blockage of data transfer will terminate the recursion. From another point of view, with a cyclic parameter link in a recursive definition, nontermination can occur only if data can pass through this cyclic parameter link infinitely in every level of recursion. If the data links in this cyclic parameter link can be linked up to form a non-cyclic connected data-link list, then no data can pass through this cyclic parameter link at a certain point in the recursion. Then, nontermination will not occur under such a situation.

Therefore, if a recursive definition has at least one cyclic parameter link in which only non-cyclic connected data-link lists can be formed, we can conclude that the evaluation of this recursive definition will terminate. We can also conclude that the cyclic parameter link is an actual exit-reaching process even without any semantic knowledge of this recursive definition. In other words, the absence of non-cyclic connected data-link lists is a sufficient indicator of nontermination. The relationship between connected data-link lists and nontermination can be more thoroughly perceived after we have also examined the case of cyclic connected data-link lists.

4.4.2.2 Cyclic Connected Data-link Lists and Nontermination

At the beginning of Section 4.4.2, we have discussed how data links can form a cyclic connected data-link list with the example from Figure 4.7. We have also seen that

a cyclic connected data-link list can be considered as the repeating segment of a connected data-link list with infinite length. In this section, we shall see how a cyclic connected data-link list is related to nontermination.

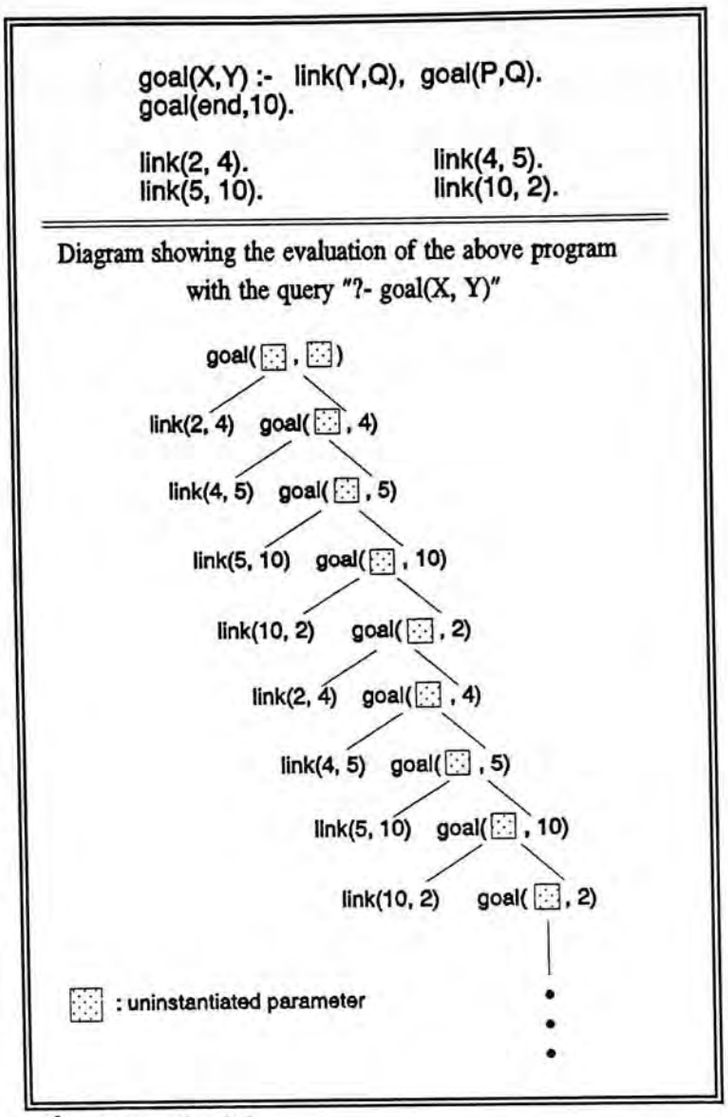


Figure 4.11

In Figure 4.11, the search tree shows a data transfer sequence that leads to nontermination. If we compare the example in Figure 4.10 with this example in Figure 4.11, we can easily recognize that it is the same recursive rule in both programs. The two programs are different only in the procedures defining the subgoal *link*. This seemingly small difference will however result in different data links, and eventually totally different kinds of connected data-link lists in the two programs. This shows us that nontermination in a pure Prolog program results from some inappropriate procedures defining the subgoals involving in the cyclic parameter link.

As mentioned in the above sections, a cyclic parameter link is a potential exit-reaching process. The subgoals involved in the cyclic parameter link can determine the values to be passed through the cyclic parameter link during the recursion by the procedures defining them while one of these subgoals can act as the exit condition at a certain point of the recursion. A subgoal acts as an exit condition because it fails at a certain point before the recursive subgoal can be evaluated. From the perspective of data transfer, a subgoal acts as an exit condition because it can stop further data transfer into another level of recursion. Nontermination occurs in a pure Prolog program if the subgoals involved in the cyclic parameter link are inappropriately defined, allowing an

infinite data transfer sequence to occur. In pure Prolog, an infinite data transfer sequence can be constructed only by repeating a common segment. The search tree in Figure 4.11 clearly shows this characteristic of the infinite data transfer sequence.

The reason for the above situation can be better understood if we consider the case in terms of data links. As shown at the beginning of Section 4.4.2, a connected data-link list is the representation of a data transfer sequence . Since a connected data-link list is constructed out of data links formed in a recursive definition, an infinite data transfer sequence without a common segment can be represented only by an infinite non-cyclic connected data-link list. If there is an infinite non-cyclic connected data-link list in a recursive definition, there must exist an infinite number of data links in the recursive definition to form the infinite non-cyclic connected data-link list. However, in pure Prolog, each data link is eventually based on one or more facts in some procedures that directly or indirectly define the subgoals involved in the cyclic parameter link in this recursive definition. An infinite number of data links can only result from an infinite number of facts in a recursive definition, that is, a recursive definition of infinite size. However, a recursive definition of infinite size is practically impossible.

Therefore, the procedures defining the subgoals that form the cyclic parameter link will lead to nontermination if they allow the data transfer sequence to have a repeating common segment. Because a cyclic connected data-link list is a representation of such a data transfer sequence, a cyclic connected data-link list is an indicator of nontermination in pure Prolog. A one-one correspondence between the infinite data transfer sequence and the cyclic connected data-link list can also be established in this case as well as the case of non-cyclic connected data-link list. On the one hand, the values appearing in the cyclic connected data-link list are also the values appearing in the corresponding data transfer sequence; on the other hand, the length of the cyclic connected data-link list can indicates the length of the repeating segment of the data transfer sequence in terms of the number of parameter cycles. In Figure 4.11, the example has data links of *2--4*, *4--5*, *5--10* and *10--2* so that a cyclic connected data-link list of *2--4--5--10--2* can be constructed. If we compare this cyclic connected data-link list with the search tree in Figure 4.11, the one-one correspondence can be clearly shown.

4.4.3 Multi-Connected Data-link Lists

Although cyclic and non-cyclic connected data-link lists are considered separately in the above sections to keep the discussion simpler, there are cases in which **more than one connected data-link lists and even more than one kinds of connected data-link lists exist in a cyclic parameter link.** Furthermore, the case of multi-connected data-link lists can occur either in a recursive definition with one cyclic parameter link or a recursive definition with multi-cyclic parameter links. Combining these possibilities together, there are many different cases of multi-connected data-link lists.

4.4.3.1 in One Cyclic Parameter Link

There are three possible cases of multi-connected data-link lists in a recursive definition with only one cyclic parameter link:

(1) all connected data-link lists are non-cyclic;

(2) all connected data-link lists are cyclic; and

(3) some connected data-link lists are cyclic and some are non-cyclic for the same cyclic parameter link.

While the multi-connected data-link lists are homogeneous in the first two cases, they are heterogeneous in the third case.

The two kinds of homogeneous connected data-link lists are significantly different from each other. This can be shown by comparing the example in Figure 4.10 with the one in Figure 4.12. In Figure 4.10, the recursive definition has some **homogeneous multi-non-cyclic connected data-link lists** for its only cyclic parameter link. In Section 4.4.1.2, we have already discussed how the backtracking mechanism plays an important role in this case of multi-non-cyclic connected data-link lists. In contrast, the backtracking mechanism does not have any effect in the case of **homogeneous multi-cyclic connected data-link lists.** This is shown by Figure 4.12. The search tree in Figure 4.12 shows why

no backtracking can happen in the case of homogeneous multi-cyclic connected data-link lists.

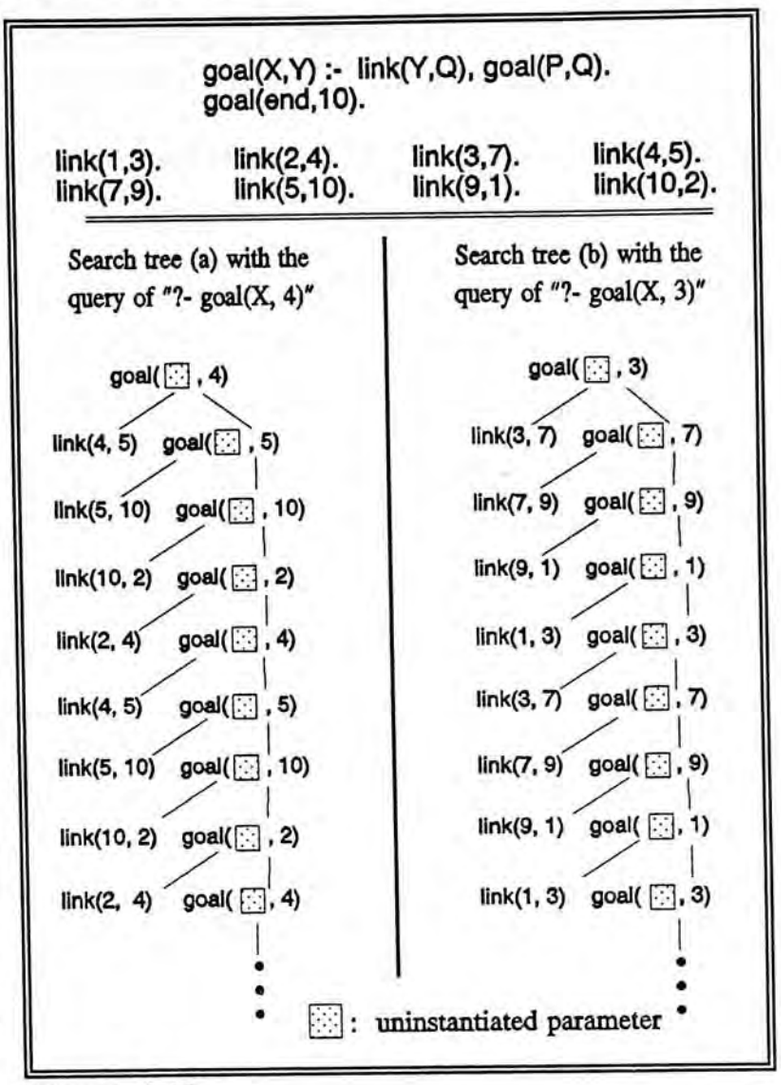


Figure 4.12

The recursive definition in Figure 4.12 is similar to the one in Figure 4.11 except with more than one cyclic connected data-link lists in for same cyclic parameter link. The two cyclic connected data-link lists are: *2--4--5--10--2*, (which also exists in the recursive definition in Figure 4.11) and *1--3--7--9--1*. If we supply different queries, different data transfer sequences may arise. In Search tree (a) in Figure 4.12, the result of the query of *?-goal(X,4)* is shown. If we examine the search tree, we can find that it is basically the same search tree as the one in Figure 4.10. With the values *2*, *4*, *5* or *10* supplied to the second parameter in the query, only the data transfer sequence corresponding to the cyclic connected data-link list of *2--4--5--10--2* can arise. On the other hand, if values *1*, *3*, *7* or *9* are used, only the data transfer sequence corresponding to the cyclic connected data-link list of *1--3--7--9--1* occurs. Search tree (b) shows the result from the query of *?-goal(X,3)*. Different from the case of multi-non-cyclic connected data-link lists, in which the backtracking mechanism allows all the possible data transfer sequences corresponding to each non-cyclic connected data-link list to be tried, no backtracking occurs in this case of multi-cyclic connected data-link list. The reason can be seen by examining these search trees. Because the data transfer sequence is infinite, the evaluation of the recursive definition following this particular sequence cannot come to an end. Backtracking occurs only at the point where a data transfer sequence comes to an end. Therefore, the nonterminating evaluation according to an

infinite data transfer sequence will not allow backtracking to take place and consequently it will not try other possible data transfer sequences.

Moreover, a particular data transfer sequence generated in the evaluation is determined by the particular value in a certain cyclic connected data-link list. In the example in Figure 4.12, since the value *4* in the query is a value in the cyclic connected data-link list of *2--4--5--10--2*, the data transfer sequence according to this connected data-link list is followed in the evaluation of this recursive definition; when the value used in the query is changed to *3*, the data transfer sequence corresponding to the cyclic connected data-link list of *1--3--7--9--1* is generated. But what will happen if no value is supplied to the parameter of the cyclic parameter link in the query, e.g., *?-goal(X,Y)*. In this case, it depends on Prolog's search strategy. As discussed in Chapter 2, Prolog searches through the procedure defining a particular subgoal to find a fact or a rule to unify with the subgoal in the recursive rule. Therefore, which data transfer sequence can be generated with the query of *?-goal(X,Y)* depends on the order of the facts that form part of a particular cyclic connected data-link list. In this case, if the searching mechanism in Prolog is from left to right, the first fact in the procedure *link* that will be encountered is *link(1,3)* which forms part of the cyclic connected data-link list of *1--3--7--9--1*. Hence, the data transfer sequence corresponding to *1--3--7--9--1* is generated.

In the above example, the two cyclic connected data-link lists are independent from each other. The values appearing in one cyclic connected data-link list are entirely different from those appearing in the other cyclic connected data-link list. However, the case of multi-cyclic connected data-link lists does not need to be the case of multiple independent cyclic connected data-link lists. For example, if the procedure *link* in the above example is modified as follows:

link(1,3).	**link(2,3).**	**link(3,7).**	**link(3,5),**
link(7,9).	link(5,10).	link(9,1).	link(10,2).

The facts in bold typeface are the ones different from those in Figure 4.12. With this modification, another two cyclic connected data-link lists, *1--3--7--9--1* and *2--3--5--10--2*, can also be constructed but they are **not independent from each other**. Both share a

common value *3*. On the other hand, instead of the above two short cyclic connected data-link lists, we can construct the long cyclic connected data-link list of *1--3--5--10--2--3--7--9--1* (or 2--3--7--9--1--3--5--10--2, they are the same). Unlike the case of the common segment in the multi-non-cyclic connected data-link lists, no backtracking can occur in the evaluation following an infinite data transfer sequence. This has already been shown by the example and its search tree in Figure 4.12. Since the two short cyclic connected data-link lists also represent two infinite data transfer sequences, the long cyclic connected data-link list cannot be the result of joining the two short ones through backtracking as the case of multi-non-cyclic connected data-link lists. Therefore, there are **two alternative methods** to construct cyclic connected data-link lists from the same set of data links if the values in one cyclic connected data-link list are not all different from the values in other cyclic connected data-link lists for the same cyclic parameter. Since each connected data-link list represents a particular data transfer sequence passing through the cyclic parameter link during the recursion, the two alternatives indicate that there are two alternative methods to evaluate the same recursive definition by following different data transfer sequences. One alternative is to follow the longer data transfer sequence corresponding to the long cyclic connected data-link list; the other alternative is to follow either one of the two shorter data transfer sequences corresponding to the two short cyclic connected data-link lists. It can be shown that the decisive factors in determining which alternative to follow are the positions of the facts in the procedures defining the subgoals involved in the cyclic parameter link and the execution model of Prolog. To simplify our discussion in this point, we shall leave the discussion of this case to the next chapter. In this chapter, we shall be concerned only with the case of independent cyclic connected data-link list.

In the case of heterogeneous multi-connected data-link lists, both cyclic and non-cyclic connected data-link lists exist in the same cyclic parameter link. At first glance, nontermination does not seem to be the inevitable result because some finite data transfer sequences indicated by the non-cyclic connected data-link lists are also present in the same cyclic parameter link. However, **because of the backtracking mechanism in Prolog,** the presence of heterogeneous multi-connected data-link lists in **a recursive definition** always implies that the evaluation of this recursive definition **will definitely end in**

nontermination. It can be illustrated by Figure 4.13. A cyclic parameter link is located at the second parameter of the recursive definition in Figure 4.13 between the parameters of *Y* and *Q*. There are two non-cyclic connected data-link lists, *0--7--8--9* and *1--2--9*, and one cyclic connected data-link list, *1--3--4--1*, that are formed from the data links *0--7*, *1--2*, *1--3*, *2--9*, *3--4*, *4--1*, *7--8* and *8--9*. By the search tree in Figure 4.13, we can clearly see that the evaluation of this program will result in nontermination although there exist two non-cyclic connected data-link lists.

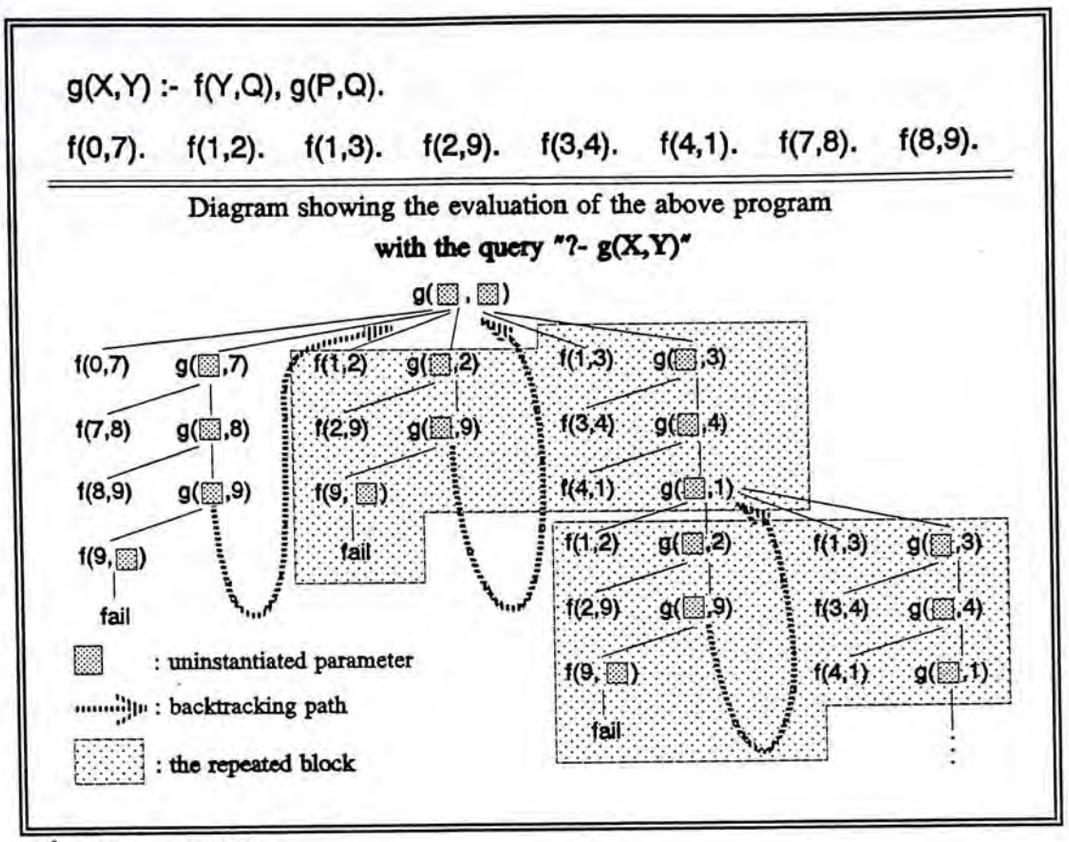


Figure 4.13

Due to the backtracking mechanism in Prolog, the evaluation will resume at other remaining possible data transfer sequences when the end of each finite data transfer sequence is reached, just as the case of multi-non-cyclic connected data-link lists shown earlier in the example and its search tree in Figure 4.10. However, in this case of heterogeneous multi-connected data-link lists, as shown by the search tree in Figure 4.13, backtracking does not only lead the evaluation to other possible finite data transfer sequences but also to the infinite data transfer sequence as well. The only data transfer sequences that cannot be reached through the backtracking mechanism are the ones that are blocked by the nonterminating evaluation following an infinite data transfer sequence. Therefore, after the finite data transfer sequences corresponding to the two non-cyclic connected data-link lists, *0--7--8--9* and *1--2--9* are tried, backtracking will

cause the evaluation of the recursive definition in Figure 4.13 to resume and to follow the infinite data transfer sequence corresponding to the cyclic connected data-link list of *1--3--4--1*. By this example, we can see that nontermination can always result from the presence of an infinite data transfer sequence although there also exist some finite data transfer sequences.

Moreover, comparing the search tree in Figure 4.13 with the one in 4.12, we can see how nonterminating evaluation can result through different ways of following an infinite data transfer sequence. In the search tree in Figure 4.12, the evaluation follows the infinite data transfer sequence closely without ever branching to other data transfer sequences. It is because the two possible cyclic connected data-link lists are independent from each other. Once one of them is followed during the recursion, the data transferred from the previous parameter cycle of recursion determines the data transferred into the next parameter cycle. In Figure 4.13, the non-cyclic connected data-link list *1--2--9* and the cyclic connected data-link list *1--3--4--1* are not independent from each other. Both share the value *1*. At a certain level of recursion where the data transferred from the previous parameter cycle is *1*, the evaluation of the recursive definition has two alternatives to follow: one is the infinite data transfer sequence and the other is the finite one. Usually, the Prolog execution model will decide which course to take in this case by the positions of the facts used to define the procedures forming the cyclic parameter link (i.e, also the procedures forming the connected data-link lists). In Figure 4.13, the finite data transfer sequence corresponding to *1--2--9* is considered to be located before the infinite one corresponding to *1--3--4--1* and therefore the finite data transfer sequence is chosen to be followed first. However, at the end of the finite data transfer sequence, backtracking takes place again and resumes the evaluation according to the infinite data transfer sequence corresponding to *1--3--4--1*. Thus, the evaluation can never escape from this nonterminating cycle. This is indicated as the repeated blocks (the shaded areas) in the search tree in Figure 4.13. In summary, **the presence of one cyclic connected data-link list in a recursive definition with only one cyclic parameter link is a sufficient indicator of nontermination.**

The discussion of the presence of multi-connected data-link lists in a cyclic parameter link is not complete if we do not consider the **evaluation order for the different data transfer sequences represented by the different connected data-link lists.** To construct a connected data-link list, a set of data links must be first constructed out of the analysis of the procedures defining the subgoals of a cyclic parameter link. According the sequence of different facts in different procedures, different data links are constructed in a certain sequence. For example, in the recursive definition in Figure 4.13, the first data link formed is *0--7* and then is followed by *1--2*, *1--3*, *2--9*, *3--4*, *4--1*, *7--8* and *8--9*. According to the sequence of these data links, connected data-link lists are also constructed following a specific sequence: *0--7--8--9*, *1--2--9* and *1--3--4--1*. Since the order of these connected data-link lists is based on of the data links, we can view that the connected data-link list of *0--7--8--9* is located before the connected data-link list of *1--2--9* while the set of *1--2--9* is located before the set of *1--3--4--1*. For each connected data-link list, there is one data transfer sequence that will be followed in the course of the evaluation of the recursive definition. As indicated in the above discussion on the effect of backtracking, another data transfer sequence is not started until one is ended. This is shown by the search trees in Figures 4.10, 4.12 and 4.13. Therefore, the relative locations of the different connected data-link lists can indicate the order of the corresponding data transfer sequences that will be followed in the course of the evaluation.

In the case of multi-non-cyclic connected data-link lists, the relative positions of the connected data-link lists has no significance since the backtracking mechanism in Prolog can guarantee all finite data transfer sequences are tried. But, in the case of multi-cyclic connected data-link lists, the relative locations of the different cyclic connected data-link lists are significant to determine which data transfer will be followed during the recursion. It is already shown by the discussion on the example in Figure 4.12. However, the locations of cyclic connected data-link lists are not significant in terms of nontermination. Although different infinite data transfer sequences may be followed, all lead to nontermination.

In the case of the mixed kinds of connected data-link lists in one cyclic parameter link, the relative locations of different connected data-link lists are also significant in determining which data transfer sequence to be followed during the course of the recursion. As shown by the search tree in Figure 4.13, the data transfer sequences corresponding to the non-cyclic connected data-link lists are evaluated first while the non-cyclic connected data-link lists have relative locations ahead of the cyclic one. However, a small modification made to the example in Figure 4.13 can change the course of evaluation completely. Consider the case with the only modification on the positions of the facts in the procedure *f* as follows:

f(1,3). f(0,7). f(1,2). f(2,9). f(3,4). f(4,1). f(7,8). f(8,9).

If we compare it to the procedure *f* in Figure 4.13, the only change is that the fact *f(1,3)* is moved to the front of the procedure. However, because of this change in the positions of the facts, the first connected data-link list that can be formed is the cyclic connected data-link list of *1--3--4--1* instead of the non-cyclic one of *0--7--8--9*. On the other hand, because of this change, the first fact that is encountered in the search for the fact to unify with the subgoal *f* in the recursive rule during the evaluation is not *f(0,7)* but *f(1,3)*. Therefore, the data transfer sequence that will be followed by the evaluation is not the one corresponding to the non-cyclic connected data-link list of *0--7--8--9* but the infinite data transfer sequence corresponding to *1--3--4--1*. It can be shown by the search tree in Figure 4.14. The locations of the connected data-link lists are important in deciding which data transfer sequence to follow during the recursion; however, as in the case of multi-cyclic connected data-link lists, they have no effect on whether a recursive definition can terminate or not. Nevertheless, the existence of an infinite data transfer sequence will eventually lead the evaluation into a nonterminating process in spite of the existence of other finite data transfer sequences for the same cyclic parameter link. Therefore, we can conclude that **the presence of at least one cyclic connected data-link list is a sufficient indicator of nontermination in the case of multi-connected data-link lists if there is only one cyclic parameter link in the recursive definition.**

4.4.3.2 in Multi-Cyclic Parameter Links

The case of multi-connected data-link lists can be found in a recursive definition with more than one cyclic parameter links as well. There are seven possible ways in which multi-connected data-link lists exist in a recursive definition with multi-cyclic parameter links:

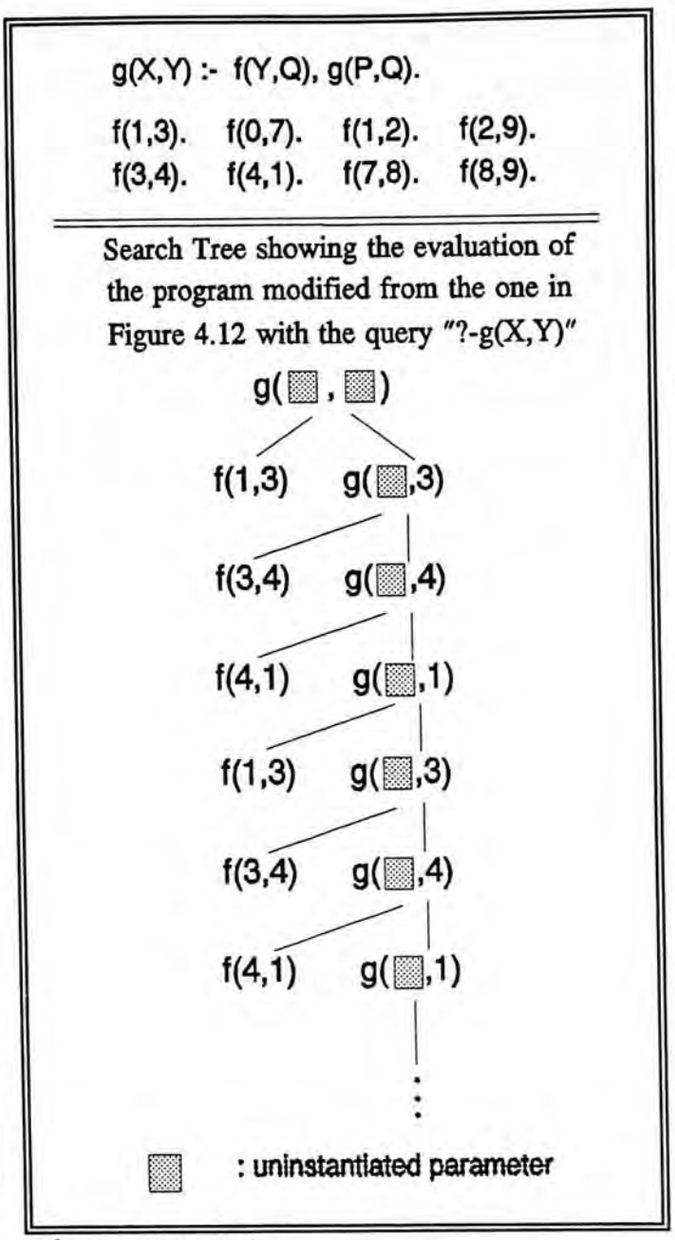


Figure 4.14

(1) only multi-non-cyclic connected data-link lists exist for all cyclic parameter links;

(2) only multi-cyclic connected data-link lists exist for all cyclic parameter links;

(3) for some cyclic parameter links, only homogeneous non-cyclic connected data-link lists exist while homogeneous cyclic connected data-link lists exist for the remaining cyclic parameter links;

(4) heterogeneous connected data-link lists exist for all cyclic parameter links;

(5) for some cyclic parameter links, only homogeneous non-cyclic connected data-link lists exist while heterogeneous connected data-link lists exist for the remaining cyclic parameter links;

(6) for some cyclic parameter links, only homogeneous cyclic connected data-link lists exist while heterogeneous connected data-link lists exist for the remaining cyclic parameter links; and

(7) for some cyclic parameter links, only homogeneous non-cyclic connected data-link lists exist; for some cyclic parameter links, only homogeneous cyclic connected data-link lists exist, and heterogeneous connected data-link lists exist for the remaining cyclic parameter links.

In **case (1)**, if only non-cyclic connected data-link lists can be constructed in all the cyclic parameter links, it is obvious that **the evaluation of this recursive definition will terminate.** Based on the concept of an exit-reaching process, if only non-cyclic connected data-link lists exist for all the cyclic parameter links in the recursive definition, each cyclic parameter link can act as an exit-reaching process itself. None of the cyclic parameter links will lead the evaluation into endless recursion. On the other hand, based on the concept of data transfer, each non-cyclic connected data-link list represents a finite data transfer sequence. The evaluation of a recursive definition that has only finite data transfer sequences to follow in all of its potential exit-reaching processes will surely terminate.

In the discussion in Section 4.4.3.1, we have found that nontermination occurs if at least one cyclic connected data-link list is present in a recursive definition with only one cyclic parameter link. Therefore, the case of homogeneous multi-cyclic connected data-link lists is no different from the case of heterogeneous connected data-link lists in terms of nontermination. In the case of multi-cyclic parameter links, cases (2), (4) and (6) have at least one cyclic connected data-link list in all of their cyclic parameter links. Since only independent cyclic parameter links are considered in this chapter, cases (2), (4) and (6) can be considered together **as the extended cases** of the case discussed in Section 4.4.3.1 (i.e., the case with at least one cyclic connected data-link list in a recursive definition with only one cyclic parameter link). This can be explained by considering the extended cases in two respects. On the one hand, because cyclic parameter links are independent from each other, the data transfer sequences for one cyclic parameter link has no effect on the data transfer sequences in other cyclic parameter links; on the other hand, as shown in the previous section, because the values passing through the cyclic parameter link with at least one cyclic connected data-link list can always form an infinite sequence due to the backtracking mechanism, the recursive definition with a cyclic parameter link in which only some connected data-link lists or all connected data-link lists are cyclic will result in nontermination. If one or more cyclic connected data-link lists exist for all the cyclic parameter links in a recursive definition, the implication is that all these cyclic parameter links cannot act as an exit-reaching process and the evaluation of this recursive definition will end in nontermination. This

can be better understood by using the data transfer analogy. With the presence of at least one infinite data transfer sequence in each independent cyclic parameter link, there will always be some values that can pass through all of the different cyclic parameter links in each level of recursion. Hence, none of these cyclic parameter links can act as an exit-reaching process. Without an exit-reaching process, a recursive definition is cannot terminate. Therefore, **nontermination occurs in cases (2), (4) and (6).**

On the other hand, in cases (3), (5) and (7), not all non-cyclic connected data-link lists exist for all of their cyclic parameter links, nor at least one cyclic connected data-link list exists in all of their cyclic parameter links. However, **some of their cyclic parameter links have only non-cyclic connected data-link lists** while **the remaining cyclic parameter links have at least one cyclic connected data-link list.** In the same recursive definition, we can consider that there are two kinds of cyclic parameter links: some of its cyclic parameter links seem to indicate the evaluation of the recursive definition can terminate (i.e., the ones with the presence of only non-cyclic connected data-link lists) while the remaining ones seem to indicate that the evaluation will end in nontermination (i.e., the ones with at least one cyclic connected data-link list). Even though these cases consist of only independent cyclic parameter links, they cannot be considered as the extended cases of any case discussed in Section 4.4.3.1 since it is not possible to have two or more different kinds of cyclic parameter links in the recursive definition with only one cyclic parameter link. In order to determine the outcome of these cases, we must study how the presence of two different, or even apparently opposite, kinds of cyclic parameter links affects the evaluation of a recursive definition.

In Figure 4.15, there is **a multi-cyclic parameter links recursive definition. There** are two independent cyclic parameter links and one of them has only **non-cyclic** connected data-link lists while the other cyclic parameter link has at **least one cyclic** connected data-link list. The result is shown by the search tree in **Figure 4.15; the** evaluation of such a recursive definition can terminate although **one of the cyclic** parameter links of the recursive definition has a cyclic connected **data-link list. The** example shows that the cyclic parameter link with a non-cyclic **connected data-link list** is more dominant in the course of evaluation than the cyclic **parameter link with a**

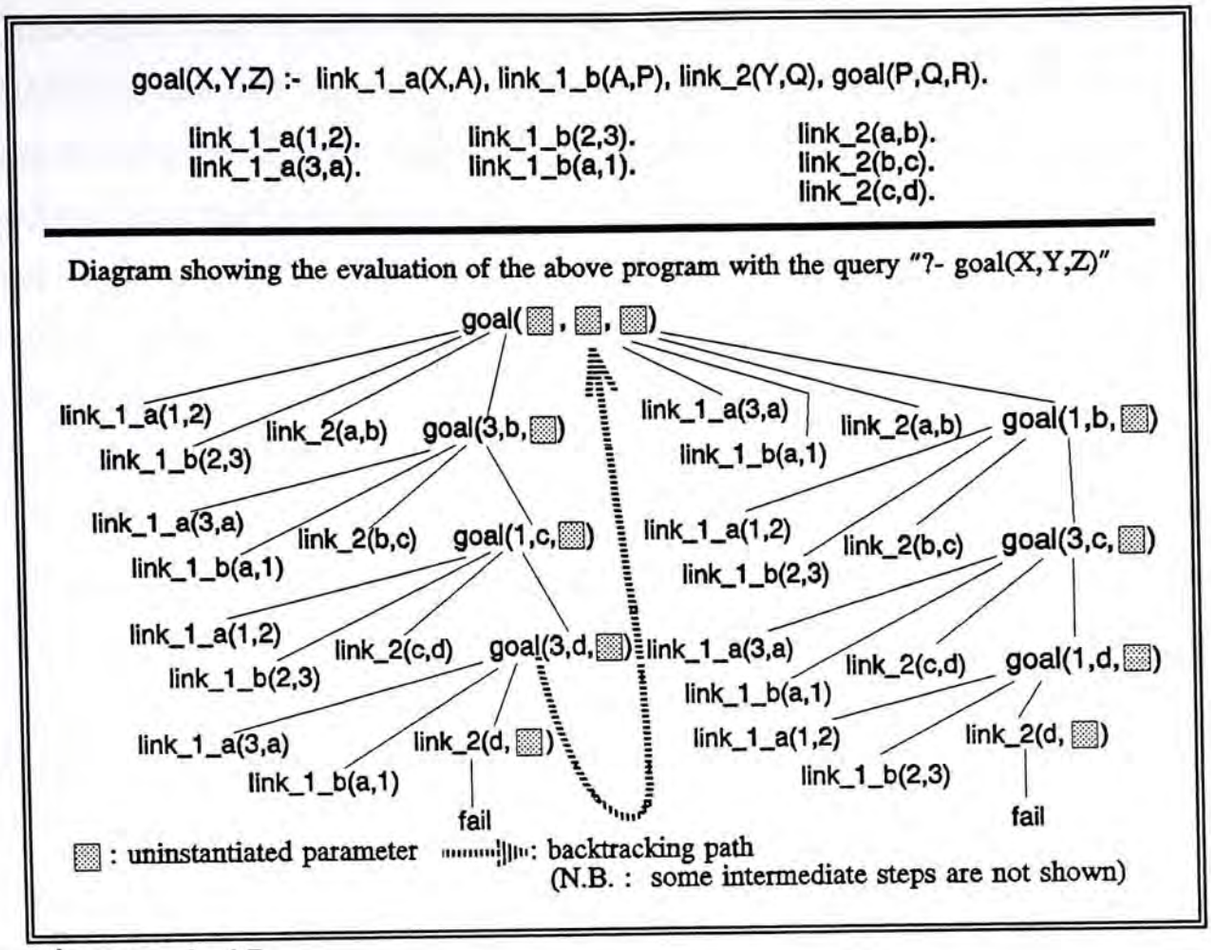


Figure 4.15

cyclic connected data-link list. By examining the search tree in Figure 4.15, we can see why the two independent cyclic parameter links can affect each other. They are independent from each other since they are formed by two different sets of subgoals so that the data transfer sequence in one of them cannot affect the sequences in the other. However, the two cyclic parameter links in the recursive definition in Figure 4.15 affect each other **not** in terms of the data transfer sequence but in terms of where the evaluation ends. Two independent cyclic parameter links can affect each other because the subgoals forming each of them are parts of the same recursive rule. If any subgoal in a rule fails, the evaluation of the whole rule fails. Therefore, the evaluation of a recursive definition can continue only as long as the evaluation of each subgoal in every cyclic parameter link in the recursive definition can succeed.

If a cyclic parameter link has **only one non-cyclic connected data-link list**, the values passing through this cyclic parameter link can only form finite data transfer sequences. One of the subgoals forming the cyclic parameter link must fail when the end

of the finite data transfer sequence is reached. In the case of **more than one non-cyclic connected data-link lists** for one cyclic parameter link, there are more than one possible data transfer sequences when values are transferred through the cyclic parameter link. The backtracking mechanism can resume evaluation at the next possible data transfer sequence once a subgoal fails at the end of the previous data transfer sequence. However, when the last possible data transfer sequence is exhausted, one subgoal in this cyclic parameter link will fail and no further backtracking within the same cyclic parameter link can occur.

The case of multi-cyclic parameter links recursive definition can also be understood in terms of an exit-reaching process. Based on the concept of an exit-reaching process, each cyclic parameter link in a recursive definition with a multi-cyclic parameter link is a potential exit-reaching process. A subgoal in a cyclic parameter link can act as an exit condition when it fails and blocks any further evaluation of the recursive definition. Therefore the subgoals forming the cyclic parameter link are potential exit conditions. In Prolog, as discussed in Chapter 2, if there are more than one possible exit condition in a recursive definition, it is only necessary to reach one of them to terminate the evaluation of the recursive definition. For a cyclic parameter link with only non-cyclic connected data-link lists, one of the subgoals forming this cyclic parameter link must fail at a certain point of the recursion and become an exit condition. Therefore this cyclic parameter link is an actual exit-reaching process. With the presence of an exit-reaching process, the recursive definition will terminate even though the other cyclic parameter link may contain infinite data transfer sequences.

As a conclusion, **nontermination occurs if all of the cyclic parameter links of any one of the recursive definitions have at least one cyclic connected data-link list in a Prolog program with only independent cyclic parameter links.** Therefore, we can detect nontermination in a pure Prolog program (if it contains only independent cyclic parameter links) by examining the presence of any cyclic connected data-link list in all of the cyclic parameter links in each of its recursive definitions.

4.4.3.3 The Case of Multiple Recursive Subgoals in the Same Rule

The last case of multi-connected data-link lists to be considered is a special case of multi-connected data-link lists. Unlike the cases discussed in Sections 4.4.3.1 and 4.4.3.2, which are the cases of multi-connected data-link lists in a recursive definition with only one recursive subgoal, the case discussed in this section is the case of multi-connected data-link lists in **multiple recursive subgoals in the same recursive rule.**

In Figure 4.16, there are two examples of this kind of special cases. For both examples, there are two recursive subgoals in the same recursive rule *in_order*. They are identical except for the fact that the positions of two recursive subgoals are interchanged. In both programs, there is a cyclic parameter link at the second parameter for both recursive subgoals. By analyzing the procedure defining the subgoal *tree*, we can find that the first recursive subgoal in Program (a) and the second recursive subgoal in Program (b) have data links *a--b*, *b--a*, *c--nil* and *d--nil* while the second recursive subgoal in Program (a) and the first recursive subgoal in Program (b) have data links *a-c*, *b--d*, *c--nil* and *d--nil*. Therefore, for the first recursive subgoal in Program (a) and the second recursive subgoal in Program (b), we can find a cyclic connected data-link list of *a--b--a* and two non-cyclic connected data-link lists of *c--nil* and *d--nil*, while the second recursive subgoal in Program (a) and the first recursive subgoal in Program (b) have only non-cyclic connected data-link lists of *a--c--nil* and *b--d--nil*. **In the same rule, there exist both a recursive subgoal that has only finite data transfer sequences, and a recursive subgoal that has at least one infinite data transfer sequence.** On the one hand, the cyclic connected data-link list in one of the two recursive subgoals, *a--b--a*, indicates that the evaluation of the corresponding recursive subgoal will not terminate. On the other hand, the presence of only non-cyclic connected data-link lists in the other recursive subgoal indicates that the evaluation of the other recursive subgoal will come to an end if it is considered in isolation. If they are located in two different recursive rules, one can conclude that nontermination will arise when the program is evaluated. As discussed in Chapter 2, nontermination in one recursive definition in a program is sufficient to cause nontermination for the whole program. However, in the examples in Figure

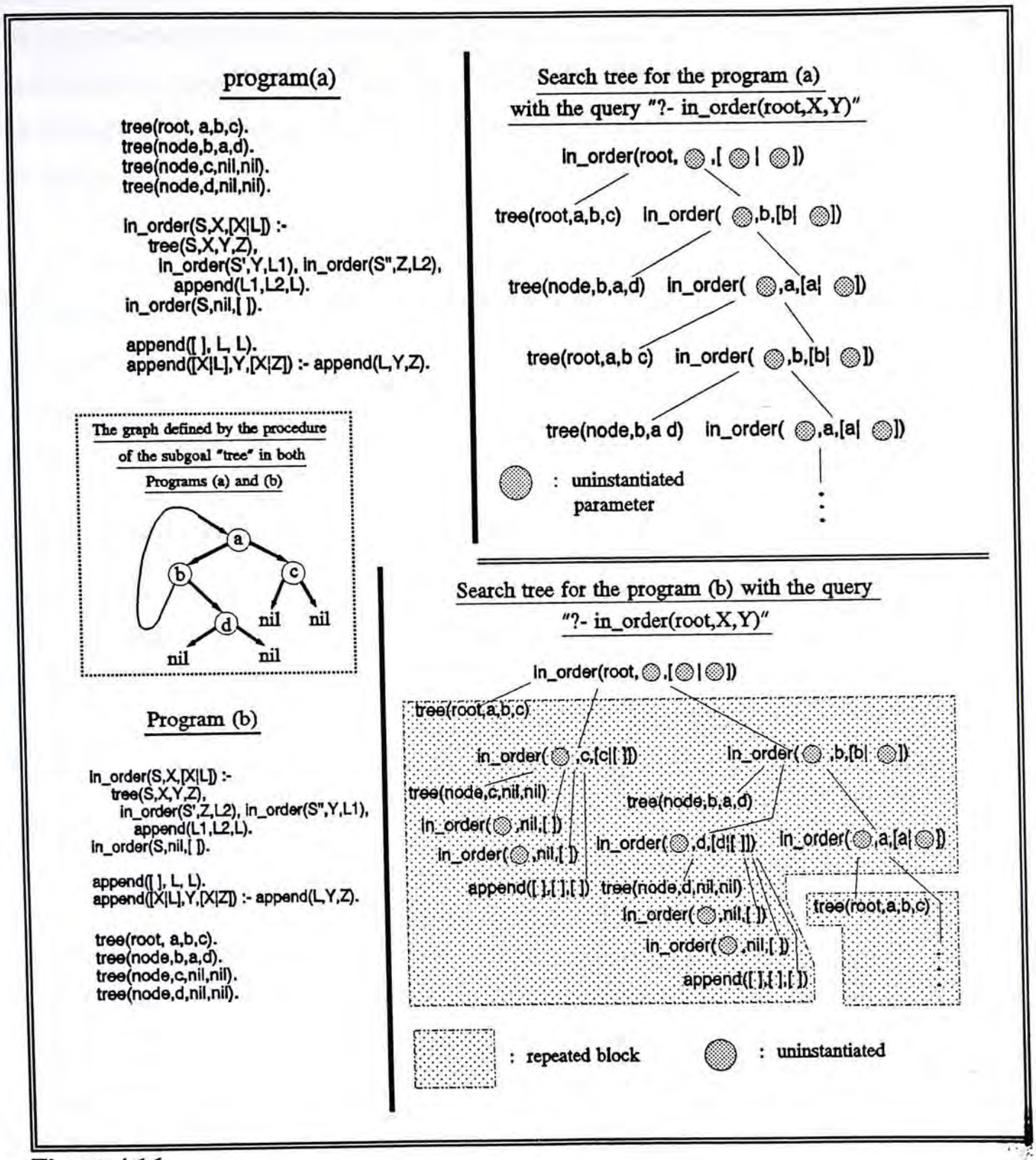


Figure 4.16

4.16, both recursive subgoals are located on the same recursive rule. As discussed in the previous section, the different subgoals in the same rule can affect the course of recursion of each other because only one subgoal is sufficient to block the course of recursion of all other subgoals in the same rule. Can the recursive subgoal with only finite data transfer sequences block the nontermination caused by the evaluation of the

recursive subgoal with some infinite data transfer sequences? We must examine how the two recursive subgoals, recursive subgoal that has only non-cyclic connected data-link lists and recursive subgoal that has at least one cyclic connected data-link list, interact with each other in the same rule before we can conclude whether nontermination will occur in such a special case.

In Program (a), for there is a cyclic connected data-link list in the only cyclic parameter link of its first recursive subgoal, the evaluation of its first recursive subgoal will not terminate. However, if the evaluation of the first recursive subgoal does not terminate, the second recursive subgoal can never be reached. Search tree (a) in Figure 4.16 clearly shows this. Although the non-cyclic connected data-link lists in the second recursive subgoal indicate that the evaluation of this second recursive subgoal will terminate if it is evaluated by itself alone, the second recursive subgoal can never be evaluated and cannot affect the termination of this program. Nontermination occurs. By examining Search tree (a), we can clearly see that only the first recursive subgoal is evaluated in a nonterminating sequence. The evaluation of the second recursive subgoal is blocked and has no effect in this case. Therefore, if there are several recursive subgoals in the same recursive rule and the nonterminating one precedes all other terminating ones, the evaluation of this recursive rule will result in nontermination.

In Program (b), we interchange the positions of these two recursive subgoals. What has been the first recursive subgoal in Program (a) now becomes the second recursive subgoal in Program (b) so that the recursive subgoal that has only finite data transfer sequences precedes the nonterminating one. However, the evaluation of the program still results in nontermination if the first recursive subgoal can terminate with its evaluation being successful. Search tree (b) in Figure 4.16 shows why the recursive subgoal that can terminate by itself cannot block the evaluation of the other recursive subgoals located behind it. Although the first recursive subgoal in Program (b), as indicated by the non-cyclic connected data-link lists, can terminate properly by itself, the execution of the whole recursive rule cannot. If any subgoal, including the recursive subgoal, succeeds in its evaluation, the next step is to evaluate the other subgoals following it one by one. Therefore, in Program (b), after the first recursive subgoal is

successfully evaluated, the next step is to evaluate the second recursive subgoal. The evaluation of the second recursive subgoal causes the recursion to go down one more level where everything that has happened before repeats again: the first recursive subgoal is evaluated again and the next level of recursion is tried for the evaluation of the second recursive subgoal. These are all shown in Search tree (b). The evaluation will not stop until the evaluation of the second recursive subgoal can stop. However, there is an infinite data transfer sequence in the second recursive subgoal. Hence, the evaluation of the second recursive subgoal cannot stop and the entire recursive rule runs into nontermination. Therefore, nontermination will arise if there exists any cyclic connected data-link list for all the cyclic parameter links of any recursive subgoal despite the number and the location of the recursive subgoal in one recursive rule.

However, Program (b) can **avoid nontermination under a special situation** if the procedure defining the subgoal *tree* is modified as follows:

tree(root,a,b,c). tree(node,b,a,d). tree(node,d,nil,nil).

By removing the fact *tree(node, c, nil, nil)*, the evaluation of the first recursive data transfer sequence in the only cyclic parameter link in the first recursive subgoal. (There is no cyclic parameter link in the first parameter of the first recursive subgoal between parameters *S* and *S'* nor in the first parameter of the second recursive subgoal between parameters *S* and *S"* because there is no subgoal nor any special parameter to link them up. The first parameter in both recursive subgoals are used to regulate the program so that there is no backtracking to the subtrees once the whole binary tree is traversed.) Because the evaluation of the first recursive subgoal fails, the first recursive subgoal blocks further evaluation of any other subgoals in the same rule. Therefore, the evaluation of Program (b) can termination but no evaluation can succeed. This is shown by Search tree (a) in Figure 4.17.

Moreover, a similar result can be obtained if we remove the fact *tree(node, d, nil, nil)* instead of the fact *tree(node, c, nil, nil)*. The result of removing the fact *tree(node, d, nil, nil)* is shown by Search tree (b) in Figure 4.17. With the presence of the fact

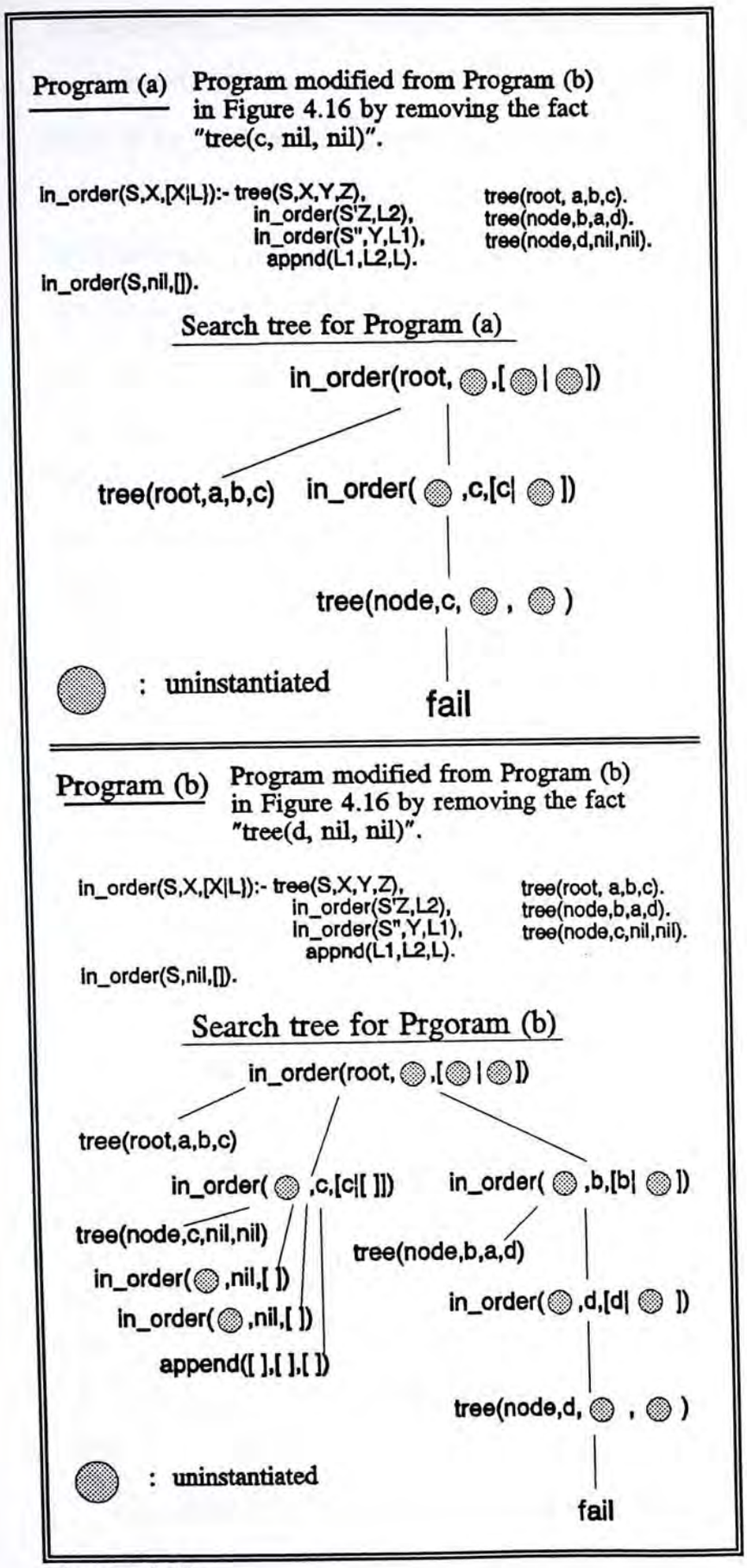


Figure 4.17

data transfer sequence in the first recursive subgoal allows the evaluation of the first recursive subgoal to succeed in the first level of *tree(node, c, nil, nil)*, the recursion. However, by removing the fact *tree(node, d, nil, nil)*, the evaluation of the first recursive subgoal in the second level of recursion fails. Since there is no other possible data transfer sequence in the first recursive subgoal, no backtracking occurs. Hence, the failure of the first recursive subgoal blocks the evaluation of the nonterminating second recursion. However, by removing the fact *tree(node, d, nil, nil)*, the evaluation of the first recursive subgoal in the second level of recursion fails. Since there is no other possible data transfer sequence in the first recursive subgoal, no

backtracking occurs. Hence, the failure of the first recursive subgoal blocks the evaluation of the nonterminating second recursive subgoal. The above discussion shows that: if any recursive subgoal in a recursive rule with multiple recursive subgoals has some infinite data transfer sequence, either the entire recursive rule becomes nonterminating due to this nonterminating recursive subgoal or the whole recursive rule terminates improperly, i.e., none of any recursive subgoal in the recursive rule can succeed in its evaluation. In general, each recursive subgoal in a recursive rule of multiple recursive subgoals needs to be evaluated successfully in order to perform any function. Therefore, the presence of any nonterminating recursive subgoal in a recursive rule with multiple recursive subgoals should be regarded as an indication of nontermination although the evaluation of the recursive rule may eventually escape from nontermination by another error: that is, the recursive subgoals in this recursive rule fail to perform any function.

4.5. Special Parameters and Data Links

In this section, we shall extend our discussion to the relationship between nontermination and connected data-link lists by considering the cases with special parameters involved. As has been pointed out in Chapter 3, apart from subgoals, lists and/or structured data can be used to form a cyclic parameter link. As discussed in the beginning of Section 4.1, if any data can be transferred through any cyclic parameter link formed by these lists and/or structured data, which are the special parameters, a data link can be established through some special parameters. Therefore, we can classify data links into three different kinds:

(1) data-links formed purely with subgoals,

(2) data-links formed with special parameters only, and

(3) data-links with both subgoals and special parameters involved.

As special parameters greatly differ from subgoals, the data links formed with only special parameters behave very differently from those formed purely by subgoals. The difference is so great that it is necessary to consider the above three cases separately. In the above sections, we are only concerned with case (1). In the following sections, we shall discuss how case (2) and (3) can be handled. Later, we shall show how the nontermination detecting strategy outlined in the above sections can be generalized to cover all the different kinds of data links.

4.5.1. Data Links with Special Parameters Only

Unlike the data link established through subgoals, the data link formed purely by special parameters does not have any procedure to be analyzed. By comparing the simple examples in Figure 4.1 and in Figure 4.2, we can see this difference. Consequently, we cannot analyze the procedure defining any subgoal to predict what are the exact data to be transferred through the connected data-link list. A new technique needs to be developed to deal with this type of data links. The key factor in the development of this new test is the **length of the lists and/or the structured data** being transferred in a data link.

To explain this technique, we would need to look back on how a cyclic parameter link is formed by the special parameters. In Program (a) in Figure 4.18, there is one proper cyclic parameter link in the first parameter of the recursive definition *goal* between the parameters *List* and *NewList* formed by lists, while an improper cyclic parameter link exists in the second parameter of the recursive definition (which is established through the common parameter *X*) in both the recursive rule head and the recursive subgoal. Since the improper cyclic parameter link does not relate to any other proper cyclic parameter link, it can be neglected in the analysis of the recursive definition. As shown by Search tree (a), this improper cyclic parameter link of common parameters cannot contribute any effect to the recursion; it is only used as a channel to transfer back the value *end* once the evaluation of the recursive definition succeeds. For the proper cyclic parameter link, although this cyclic parameter link seems to be formed by the subgoal *cut_one*, data link cannot be established by

Program (a)

Recursive definition
with data links formed by special
parameters that can terminate

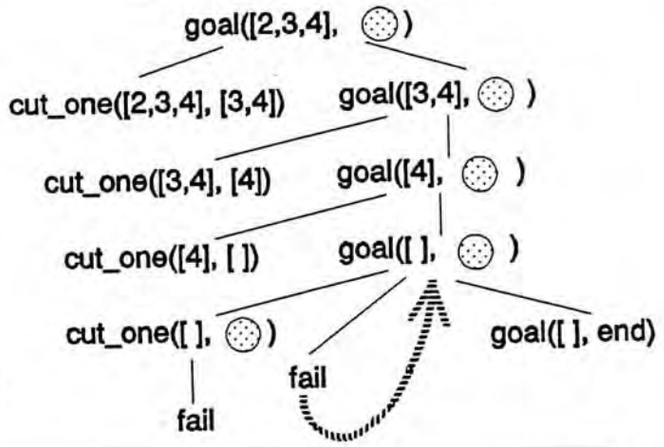
```
goal(List, X) :- cut_one(List,NewList),
                 goal(NewList, X).
goal([], end).

cut_one([X|ListRemain], ListRemain).
```

: backtracking route

: uninstantiated parameter

Search tree (a)

Search tree for Program (a)
with the query of "?- goal([2,3,4],X)"



Program (b)

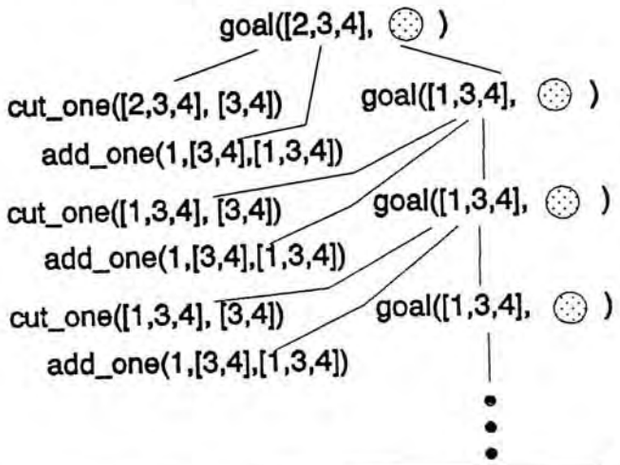
Nonterminating recursive definition with
data links formed by special parameters
that has data of constant length
during the recursion

```
goal(List, X) :-
  cut_one(List,NewList),
    add_one(1,NewList,NewList'),
      goal(NewList', X).
goal([], end).

cut_one([X|ListRemain], ListRemain).

add_one(X,List,[X|List]).
```

: uninstantiated parameter

Search tree (b)

Search tree for Program (b)
with the query of "?- goal([2,3,4],X)"



Program (c)

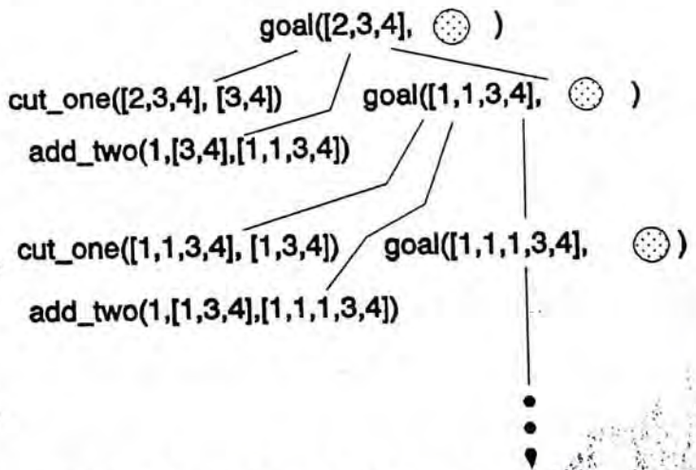
Nonterminating recursive definition with
data links formed by special parameters
that has data of increasing length
during the recursion

```
goal(List, X) :-
  cut_one(List,NList),
    add_two(1,NList,NewList),
      goal(NewList, X).
goal([], end).

cut_one([X|ListRemain], ListRemain).

add_one(X,List,[X|List]).

add_two(X,List,NewList) :-
        add_one(X,List,NList).
          add_one(X,NList,NewList).
```

: uninstantiated parameter

Search tree (c)

Search tree for Program (c)
with the query of "?- goal([2,3,4],X)"



(N.B. : The intermediate steps for the evaluation of the subgoal "add_two" are not shown.)

Figure 4.18

analyzing the procedure of this subgoal in the manner described in the above sections because the only fact defining the subgoal is *cut_one([X|ListRemain], ListRemain)* in which all its arguments have no specific value. If Program (a) in Figure 4.18 is compared to Program (a) in Figure 4.2, we can easily recognize that, although a subgoal is involved in forming the cyclic parameter link of Program (a) in Figure 4.18, the cyclic parameter link in Program (a) in Figure 4.18 is not a cyclic parameter link of subgoals. Instead, both examples belong to the same kind: they both form cyclic parameter links and data links only through some special parameters. By analyzing the procedure defining the subgoal *cut_one*, one can be sure about three things: first, since the subgoal *cut_one(List, NewList)* is defined only by the fact *cut_one([X|ListRemain], ListRemain)*, the parameters *List* and *NewList* can only be instantiated with lists; second, the second parameter is always a result of taking away the first element of the list in the first parameter; third, the subgoal *cut_one* will become fail if any non-list data or an empty list is passed to the first parameter. In other words, as long as non-empty lists are used, an infinite number of data links can be formed between the parameters *List* and *NewList*. This is the same conclusion obtained in the discussion about the examples in Figure 4.2 in the beginning of Section 4.1. In this situation, we cannot detect cyclic connected data-link lists with the method discussed above because we cannot obtain a specific set of data links and so it is not possible to link the data links to form any connected data-link lists for further analysis. However, this problem can be solved if we can find other indicators for detecting infinite data transfer sequences. **The length of the list transferred during the recursion is an indicator of infinite data transfer sequences in the case of special parameters.** This can be clearly shown by comparing Programs (a), (b) and (c) and their corresponding search trees in Figure 4.18.

As mentioned above, in Program (a), data can be transferred through its cyclic parameter link into the next level of recursion as long as the data transferred are not empty lists. On the other hand, because of the subgoal *cut_one*, the list that can pass into the next level of recursion through the cyclic parameter link must be a list having the length of one element shorter than the list passed from the previous level of recursion. Therefore, **during the recursion, the list passing through becomes shorter**

and shorter and eventually becomes an empty list. At this point, the subgoal *cut_one* fails and stop further recursion. Although **the fact *goal([], end)*** will provide the solution for the parameter *X* in the query eventually, it **is not the exit condition** to stop further recursion along the recursive rule. It is reached by the backtracking mechanism after the recursion is terminated by the subgoal *cut_one*. Search tree (a) also illustrates this clearly. Therefore, the subgoal *cut_one* acts as an exit condition at this point and the cyclic parameter link established through the subgoal *cut_one* actually performs as an exit-reaching process. Hence, there exists a finite data transfer sequence in this cyclic parameter link although we cannot construct a non-cyclic connected data-link list as in the case of cyclic parameter links involving only subgoals. By examining Search tree (a), we can find that the data transfer sequence passing through the cyclic parameter link between *List* and *NewList* during the evaluation of the recursive definition with the query *?- goal([2, 3,4], X)* is *[2,3,4], [3,4], [4], []*. With a list of three elements long, the length of the data transfer sequence is also three. If we compare Search tree (a) with the recursive definition in Program (a), we can see why there exists a correspondence between the length of the data transfer sequence and the length of the list. Since the list will be reduced by one element in every level of recursion and will block the recursion once it becomes empty, the levels of recursion that can occur is equal to the number of elements in the list. Since any list that can be supplied in the query will always be a list of finite length, the number of recursions that can occur is also finite. In this way, no nontermination occurs in Program (a). However, not all recursive definitions with cyclic parameter links formed by lists can have its lists reduced in length during the recursion. In those cases, nontermination will occur.

In Program (b), there is a recursive definition similar to the one in Program (a) except that the subgoal *add_one* is added to the cyclic parameter link. However, due to the fact *add_one(X, List, [X|List])* used to define the subgoal, any data passing through the cyclic parameter link established through the second and the third argument of the subgoal will have its length increased by one element. Therefore, any non-empty list passing through the cyclic parameter link in Program (b) will first have one element cut off from its head and then, according to the subgoal *add_one(1, NewList, NewList')*, the element *1* is added to it at its head. The length of the list transferred to the next level

of recursion will be equal to the list passed from the previous level . And the length of the list remains the same during the recursion. As indicated in Program (a), the subgoal *cut_one* can act as an exit condition only when an empty list is encountered during the recursion and therefore the cyclic parameter link can work as an exit-reaching process only if the list passing through can become shorter during the evaluation of the recursive definition. But the list passing through the cyclic parameter link always has the same length. The exit condition can never be reached and nontermination occurs. By comparing Search tree (b) with Search tree (a), it is easy to figure out how the subgoal *add_one* cancels out the effect of the subgoal *cut_one* to result in nontermination.

For Program (c), the subgoal *add_two* replaces the sub- goal *add_one* in Program (b) to form a cyclic parameter link in the recursive definition. When we examine the procedure defining the subgoal *add_two*, it is not hard to find that it can add two elements to the head of any list passing through its second and third parameter. Therefore, any non-empty list passing through the cyclic parameter link in Program (c) will have its first element cut off first and then have two elements added at the head of the remaining list. In terms of the length of the list, any non-empty list will gain one element in length when one level of recursion is completed. Instead of having a shorter and shorter length, the length of the list passing through the recursive definition increases continuously during the recursion. This can be shown by Search tree (c) in Figure 4.18. Therefore, the exit condition can never be met and nontermination also occurs just as what is indicated by Search tree (c).

In the three examples of Figure 4.18, only lists are used to illustrate how the length of the data can be an indicator of the data links formed by special parameters. However, the same is true for structured data, which represent another case of special parameters. Actually, we can give examples of structured data similar to those given in Figure 4.18. In Figure 4.19, the subgoal *cut_one* has a similar function as its counterpart in Figure 4.18. However, instead of taking away the first element of a list, *cut_one* in Figure 4.19 cuts away the functor of any structured data passing through

Program (a)
Recursive definition with data links formed by special parameters that can terminate.

```
goal(SData, X) :- cut_one(SData, NewSData),
                  goal(NewSData, X).
goal(f(), end).

cut_one(f(X), X).
```

Program (b)
Nonterminating recursive definition with data links formed by special parameters that has data of constant length during the recursion.

```
goal(SData, X) :- cut_one(SData, NewSData),
                  add_one(NewSData, NewSData'),
                    goal(NewSData', X).
goal(f(), end).

cut_one(f(X), X).          add_one(X, f(X)).
```

Program (c)
Nonterminating recursive definition with data links formed by special parameters that has data of increasing length during the recursion.

```
goal(SData, X) :- cut_one(SData, NewSData),
                  add_two(NewSData, NewSData'),
                    goal(NewSData', X).
goal(f(), end).

cut_one(f(X), X).          add_one(X, f(X)).

add_two(X, f(Y)) :- add_one(X, Y).
```

Figure 4.19

the subgoal. Therefore, if the query *?- goal(f(f(f(f()))), X)* is supplied, the data transfer sequence for the first parameter of the recursive definition is *f(f(f(f()))), f(f(f())), f(f()), f()*. In a similar manner, *add_one* adds one more functor to any structured data passing through the subgoal while *add_two* adds two more. Comparing these examples with those in Figure 4.18, it can be easily seen that the exit condition is *cut_one(SData, NewSData)*, which fails inevitably when the structured data *f()* is supplied to its first argument, blocking further data transfer to the next level of recursion. In Programs (b)

and (c) in Figure 4.19, this exit condition can never be reached since the structured data can never become *f()*.

By examining these examples in Figures 4.18 and 4.19, we draw this conclusion: if we have a way to analyze the length of the data during the evaluation of the recursive definition, we can detect any infinite data transfer sequence in the case of special parameters although we cannot construct connected data-link lists as in the case of subgoals. By analyzing the operations performed on the special parameters in these examples, we can analyze the cyclic parameter links to determine the tendency of the change in the length of the data passing through these cyclic parameter links during the recursion. If lists are involved in a data link, either it involves subgoals defined by facts whose arguments are lists, or some list operators are involved, like *head-tail separator*, "|", used in the examples in Figure 4.18. If it is the first case, it is no different from the data links formed by subgoals and can be treated in the same way. It is in the second case that we cannot use the method developed in the above sections to determine what the exact data links are.

However, for a data link or partial data link that can be established with the use of *head-tail separator*, **the data transferred to other parameters must be part of the original list.** For example, in Figure 4.18, the fact *cut_one([X|ListRemain], ListRemain)* allows data links to be formed because the tail of the list in the first argument of the subgoal *cut_one* can be transferred to its second argument. If we change the fact to be *cut_one([X|List], ListRemain)*, no data link can be considered to exist because *List* and *ListRemain* are two independent variables. The values that can be instantiated to the variable *List* bears no relation to the values that can be instantiated to the variable *ListRemain*. Such a modification breaks down any dependency between the first argument and the second argument of the subgoal *cut_one*. Therefore no data link can be established if the fact is changed in this way. In order to complete a data link, part of the original list must be passed to other parameters in the cyclic parameter link. Therefore, **there always exists a difference in the length of the original list and the length of the remaining part transferred to other parameters**, and the difference in length between the data that can be instantiated to

these two parameters is the same in every level of recursion. For Program (a) in Figure 4.18, the list that is transferred to the parameter *NewList* always has one element shorter than its original list transferred to the parameter *List*. If no other factor operates to increase the length of the remaining list before it is transferred into the next level of recursion, as in the case of Program (a) in Figure 4.18, the list becomes shorter and shorter during the recursion and is eventually reduced to an empty list. However, the mechanism that cuts the list short, in this case, the head-tail separator, will fail when an empty list is encountered. It blocks further recursion along the same recursive rule. At this point, the data transfer sequence comes to an end. However, since the head-tail separator is a standard operator in Prolog and its behavior is fully defined, the relation between the length of the original list and its remaining part can be expressed by a formula as follows:

$$\mathbf{M} = \mathbf{N} + \mathbf{1}$$

where **M** is the length of the original list while **N** is the length of the remaining part. Therefore, for Program (a) in Figure 4.18, the length of the list passed from the previous level and the length of the list passed into the next level of recursion has the relation:

length of list to next level = length of list from previous level - **1.**

For Program (b) in Figure 4.18, the relation of the length of the data transfer between the parameters *List* and *NewList* through the subgoal *cut_one* can be expressed as:

length of **NewList** = length of **List** - 1

while the relation between the length of the data passing through the parameters *NewList* and *NewList'* through the subgoal *add_one* is:

length of **NewList'** = length of **NewList** + 1.

Therefore, the relation of the data passing through the parameters *List* and *NewList'*, that is, the relation between the list from the previous level of recursion and the list to the next level of recursion, is:

length of **NewList'** = length of **List** - 1 + **1.**

This implies that the length of the list does not change during the recursion. If we apply the same analysis to Program (c) in Figure 4.18, the relation between the list passed in and the list passed out can be expressed as:

length of **NewList** = length of **List** - 1 + 1 + 1

Thus, the list passed to the next level of recursion has one more element than the list passed from the previous level. Since finite data transfer sequence can be obtained only if the list passing through the recursion can become shorter and shorter. The last two cases indicate the presence of some infinite data transfer sequences.

Therefore, if a data link is formed through lists with the presence of a head-tail separator as in the examples in Figure 4.18, we can know whether the list is growing or shrinking during the recursion by analyzing which parameter is responsible for the *tail* part of a list and where the *tail* goes to. If the *tail* directly or indirectly goes to the parameter in the recursive subgoal, we can be sure that the list passing through the recursion will be shrinking. If it goes to other special parameters that can also manipulate lists through a head-tail separator as in Program (b) and (c) in Figure 4.18, we can calculate the net difference in length between the list from the previous level of recursion and the list to the next level of recursion, to determine whether the list is growing, shrinking or remaining constant during the evaluation of the recursive definition. **This is also true for the data links formed by structured data.** For Program (a) in Figure 4.19, the fact *cut_one(f(X), X)* defining the subgoal *cut_one(SData, NewSData)* causes the most outer functor *f* to be removed from the data transferred from the parameter *SData* to the parameter *NewSData*. This can also be considered as reducing one element from the length of the structured data passing to the next level of recursion. **Moreover, a data link can also be established through lists without a head-tail separator.** We can introduce a small change to Program (a) in Figure 4.18 to remove the head-tail separator as follows:

```
goal(List, X) :-  cut_one(List, NewList), goal(NewList,X).
goal([], end).

cut_one([X,ListRemain], ListRemain).
```

A similar result will be obtained if the query to be supplied is modified as *?- goal([2, [3, [4, []]]],X)*. In this case, the list that can be passed through the recursion must be a list of lists with only two elements and the second element must be a list too. However, we can also be sure about how the length of the list changes during the recursion. There are many more possible ways to cut a list or structured data than we can consider. For example, we can have one parameter defined unusually as *[X,Y,Z|L]*. However, we can still analyze it and know the difference between *[X,Y,Z|L]* and *L* to be three elements in length. Without the need to know whether lists or structured data are transferred through these cyclic parameter links of special parameters, we can always find out the difference in length between one parameter and the other parameter in one level of recursion because the operators such as "|","," or the operation that takes away the functor of a structured data can **only increase or reduce a fixed amount of length** from the data transferred through the two parameters. Since all the rules and facts cannot be modified during the evaluation of a pure Prolog program, we can always be sure that the change in length in one level of recursion can be accumulated to form either a finite or an infinite data transfer sequence. This provides us with a way to detect the presence of infinite data transfer sequences for the cyclic parameter link established through special parameters in which the detection of connected data-link list cannot be done.

The method to detect infinite data transfer sequence in a cyclic parameter link established through special parameters can be briefly described as follows:

(1) Trace the cyclic parameter link. Identify the pair(s) of parameters P_i and P_j, where only part of the list in P_i is transferred to P_j.

(2) Evaluate the difference of length for each pair: if P_i is the parameter responsible for receiving data from the previous level of recursion, the difference is a negative value; if P_i is responsible for sending data to the next level, the difference is a positive value.

(3) Add up the differences of all pairs in the cyclic parameter link.

(4) If the result is zero or positive, the data transfer sequence is infinite; otherwise, the data transfer sequence is finite.

For example, if we modify the fact *cut_one* in Program (a) to *cut_one(ListRemain, [X|ListRemain])*, the cyclic parameter link goes from *ListRemain* to *[X|ListRemain]*. The result of the analysis described above shows a positive value, so an infinite data transfer sequence is present in this cyclic parameter link. If the query *?- goal([2,3,4],X)* is supplied again, the data transfer sequence will be *[2, 3, 4], [X 2, 3, 4], [X, X, 2, 3, 4], [X, X, X, 2, 3, 4]*, ... where *X* is the uninstantiated parameter. For Program (b), the result is zero and for Program (c), the result is positive. This shows that all of them have infinite data transfer sequences for their cyclic parameter link.

4.5.2 Data Links with Both Special Parameters and Subgoals

To form a data link with both special parameters and subgoals, the corresponding cyclic parameter link must also be established through some subgoals and some special parameters. Therefore, if we break down a data link into its partial data links, we can find that some of them are established through special parameters while other partial data links are formed by subgoals. Although the types of data transferred through each partial data link do not need to be the same, **the parameter that connects two partial data links,** i.e, the

Program (a)
Recursive definition with data links of both subgoals and special parameters

```
goal(List, X) :- link1(List, NewList),
                 link2(NewList, Value),
                       goal(Value, X).
goal([], end).

link1([X|ListRemain], ListRemain).

link2([2,3,4], 5).          link2([], 6).
```

Program (b)
Recursive definition without any data link

```
goal(List, X) :- link1(List, NewList),
                 link2(NewList, Value),
                       goal(Value, X).
goal([], end).

link1([X|ListRemain], ListRemain).

link2(3, 5).          link2(4, 6).
```

Figure 4.20

parameter being shared by the two partial data links, **must have the same kind of data.** In Figure 4.20, there are data links of both subgoals and special parameters in Program (a). The subgoal *link1* can only form partial data links through special parameters while the procedure defining the subgoal *link2* provides two partial data links of *[2, 3, 4]--5* and *[]--6*. The two partial data links can be linked together because the common parameter of the two partial data links can be instantiated with the same kind of data. If we compare Program (a) with Program (b), we can clearly see the importance of having the same kind of data for the common parameter. In spite of the great similarity between the two recursive definitions, no data link can be formed out of the partial data links because the types of data of the two partial data links are no longer compatible. The change in the facts in the procedure defining the subgoal *link2* destroys the compatibility of data type between the two partial data links.

Moreover, the kind of data that can pass into the next level of recursion is important in determining whether a data transfer sequence can go beyond one level of recursion. For example, the recursive definition in Program (a) in Figure 4.20 does not have a data transfer sequence that is longer than one level of recursion. Due to the second partial data link established by the subgoal *link2*, the data that can be transferred into the next level of recursion is, not a list, but just a value. But the first argument of the subgoal *link1* always demands a list. Therefore, the subgoal *link1* will fail at the second level of recursion and blocks further recursion along the same recursive rule. Moreover, we can understand the case as a mismatching of the type of data supposed to be received and the type of data actually sent. On the one hand, since *List*, the first parameter of the subgoal *link1* (which forms a partial data link through the special parameter of list), is also the parameter that receives data from the previous level of recursion for this cyclic parameter link, the type of data that can be received from the previous level of recursion must be lists. On the other hand, data are transferred into the next level of recursion through the parameter *Value* which is also the second parameter of the subgoal *link2*. However, according to the procedure defining the subgoal, the type of data that can be instantiated to the parameter must be some simple values of *5* and *6* but not a list. This implies that the type of data that can be transferred to the next level of recursion cannot match the type of data that is

supposed to be received from the previous level. In other words, in order to have a data transfer sequence to go beyond one level of recursion, **the type of data that can be transferred to the next level of recursion and the type of data that can be received from the previous level must be the same.** This condition required for the formation of connected data-link lists for a cyclic parameter link formed by both subgoals and special parameters is actually no different from the condition required for the formation of data links out of partial data links. For the latter case, the type of data must be the same for the common parameters of two partial data links to allow data to pass from one partial data link to another. For the former case, the parameters that are responsible for passing data from one level of recursion to the other must have the same type of data. Hence, Program (a) in Figure 4.20 can be modified, resulting in the recursive definition in Figure 4.21, which has data transfer sequences that can go beyond one level of recursion. The only modification needed is to change values *5* and *6* in the procedure defining *link2* to the lists *[5]* and *[6]*.

```
goal(List, X) :- link1(List, NewList),
                 link2(NewList, Value),
                     goal(Value, X).
goal([], end).

link1([X|ListRemain], ListRemain).

link2([2,3,4], [5]).      link2([], [6]).
```

Figure 4.21

Moreover, the recursive definition in Figure 4.21 provides an example of how an infinite data transfer sequence can be formed from data links which are formed by both subgoals and special parameters. If the query *?- goal([1, 2, 3 ,4], X)* (or any query with its first argument being four elements long with the last three elements being *2*, *3* and *4*) is supplied, nontermination occurs and an infinite data transfer sequence is formed in the cyclic parameter link as *[1,2,3,4], [5], [6], [6], [6], ...* An examination of the data transfer sequence shows that the repeating segment is essential to the formation of an infinite data transfer sequence. The relationship between the repeating segment and the infinite data transfer sequence can be shown more clearly if we change the procedure of the subgoal *link2* as follows:

```
link2([2,3,4], [4,5]).        link2([5], [a,b,c,d,e]).
link2([b,c,d,e], [a,2,3,4]).
```

and evaluate the recursive definition with the query of *?- goal([1,2,3,4], X)*. We shall get an infinite data transfer sequence of *[1,2,3,4], [4,5], [a,b,c,d,e], [a,2,3,4], [4,5], [a,b,c,d,e], [a,2,3,4], [4,5], [a,b,c,d,e], [a,2,3,4], ...* On the one hand, it does not show any similarity to the infinite data transfer sequence from the data links formed by only special parameters. For those data links formed by special parameters alone, the length of the data transferred is the indicator for detecting an infinite sequence. However, the infinite data transfer sequence in the case of data link of both subgoals and special parameters does not show the same regularity in the length of the data transferred as what has been shown in the case of data links formed by special parameters alone. On the other hand, we cannot directly construct a connected data-link list (as in the pure subgoals case) to find the data transfer sequence if both special parameters and subgoals are used to form some data links. However, the examples in Figures 4.20 and 4.21 show that **the infinite data transfer sequence in the case of data links of both special parameters and subgoals must appear in the form of some repeating identical segments.** A repeating segment therefore can be an indicator of an infinite data transfer sequence in this case.

If we consider the concept of the cyclic connected data-link list again, we shall note that a cyclic connected data-link list actually represents an infinite data transfer sequence formed out of a repeating segment. When the repeating segment in the case of data links of only subgoals is compared with the repeating segment in the case of data links of both special parameters and subgoals, we can find that the repeating segments in both cases have the same nature. Since the fact defining the subgoal *link1* in Figure 4.21 implies *List* has one more element at the head of the list than *NewList*, while *NewList* must be able to be instantiated with the values specified by the procedure defining the subgoal *link2*, we can assume that the partial data links between the parameters *List* and *NewList* are:

[X, 2, 3, 4]--[2, 3, 4] and *[X]--[]*,

where *X* is any uninstantiated parameter. On the other hand, the partial data links between the parameters *NewList* and *Value* can be concluded from the analysis of the procedure of the subgoal *link2*. They are:

[2, 3 ,4]--[5] and *[]--[6].*

If we try to connect these partial data links, we have two data links:

[X, 2, 3, 4]--[5] and *[X]--[6].*

Because the uninstantiated parameter *X* can be instantiated with the value *5*, using a data transfer analogy, the value *5* can be considered to be transferred from one data link to the next one. In other words, the list *[5]* can be considered to be equivalent to the list *[X]* when the data links are connected to form a connected data-link list *[X,2,3,4]--[5]--[6].* Moreover, the data link *[X]--[6]* can be linked up to itself because of the same reason. Therefore, an infinite connected data-link list of *[X,2,3,4]--[5]--[6]--[6]--[6]--...* can be formed. The correspondence between this infinite connected data-link list and the infinite data transfer sequence shown above is obvious. The correspondence is even more obvious in the case where the procedure of the subgoal *link2* is modified. In that case, *link2* is defined by three facts: *link2([2,3,4], [4, 5])*, *link2([5],[a,b, c,d,e])* and *link2([b, c,d,e], [a,2,3,4])*. So, the partial data links that can be formed by this subgoal are: *[2,3, 4]--[4,5]*, *[5]--[a,b, c,d,e]* and *[b,c,d,e]--[a,2,3,4])*.

Example of how connected data-link set can be formed in the case of data links of both special parameters and subgoals with using the operator "," in lists

```
goal(List, X) :- link1(List, NewList),
                 link2(NewList, Value),
                     goal(Value, X).
goal([], end).

link1([X,ListRemain], ListRemain).

link2([2,3,4], [4,[5]]).
link2([5], [a,[b,c,d,e]]).
link2([b,c,d,e], [a,[2,3,4]]).
```

```
Data Links :-  [X,[2,3,4]]--[4,[5]]
               [X,[5]]--[a,[b,c,d,e]]
               [X,[b,c,d,e]]--[a,[2,3,4]]

Linked Data Link Set :-

    [X,[2,3,4]]--[4,[5]]--[a,[b,c,d,e]]--
    [a,[2,3,4]]--[4,[5]]-- . . .
```

Figure 4.22

For the partial data links formed by the subgoal *link1*, due to the facts in the procedure of *link2*, we can consider that there exist **three** partial data **links**: *[X,2,3,4]--[2,3, 4]*, *[X,5]--[5]* and *[X,b,c,d,e]-[b,c, d,e]*. Three data links can be formed: *[X,2, 3,4]--[4,5]*, *[X,5]--*

[a,b,c,d,e] and *[X,b, c,d,e]--[a,2,3, 4]*. Because the uninstantiated parameter *X* can be instantiated with any value, lists *[X,5]*, *[X,b,c,d,e]* and *[X,2,3,4]* can be instantiated as *[4,5]*, *[a,b,c,d,e]* and *[a,2,3,4]* respectively. Therefore, an infinite connected data-link lists, *[X,2,3,4]--[4,5]--[a,b,c,d,e]--[a,2,3,4]--[4,5]--[a,b,c,d, e]--[a,2,3,4]--...* can be obtained.

Example showing how a connected data-link set can be formed from data-links of both special parameters and subgoals using Structured Data

```
goal(SData, X) :- link1(SData, NewSData), link2(NewSData, ModiSDa-
ta),
                  goal(ModiSData, X).
goal(f(), end).

link1(Y(X), X).                    link2(f(2,3,4), g(f(5))).
                                   link2(f(5), h(f(b,c,d,e))).
                                   link2(f(b,c,d,e), i(f(2,3,4))).
```

Data Links :- `Y(f(2,3,4))--g(f(5))  Y(f(5))--h(f(b,c,d,e))`
`Y(f(b,c,d,e)--i(f(2,3,4))`

Connected Data Link Set :-
`Y(f(2,3,4))--g(f(5))--h(f(b,c,d,e))--`
`i(f(2,3,4))--g(f(5))-- . . .`

Figure 4.23

Moreover, the above discussion can be extended to the case of operator "," in lists and the case of structured data. This can be shown by the examples in Figure 4.22 and Figure 4.23. By recognizing that the uninstantiated parameter can be considered to be connected to any value, list or structured data, we can establish a connected data-link list as in the case of data links of only subgoals. **The appearance of repeating segment** shows that the connected data-link list formed has infinite length (just like the cyclic connected data-link list formed in the case of data links of only subgoals). Therefore, the appearance of a repeating segment indicates that the corresponding cyclic parameter link contains an infinite data transfer sequence, and so it cannot function as an exit-reaching process.

We have compared the case of data links involving both special parameters and subgoals with the case of data links involving only special parameters or only subgoals. The former case should not be confused with the latter two cases. In the latter cases, we can analyze different types of cyclic parameter links with different methods, either by constructing connected data-link lists or by analyzing the tendency of the length of the data transfer sequence, to determine which cyclic parameter link contains infinite data transfer sequences. As discussed in Section 4.4.3, if all the cyclic parameter links contain at least one infinite data transfer sequence, we know that nontermination can occur. In the former case, the result of the analysis is only limited to the cyclic parameter link formed by both special parameters and subgoals. However, it is necessary to examine all other cyclic parameter links in the same recursive definition before we can draw any conclusion about whether this recursive definition will terminate.

4.6 Data Links and Infinite Data Transfer Sequence Detection

In this section, we shall describe how a systematic approach can be developed to detect a data link and in turn an infinite data transfer sequence. In the case of data links of only subgoals, it is a question of how to systematically construct a cyclic connected data-link list out of the data links. In the case of data links of only special parameters, it is a question of how to detect the change tendency of the length of the data being transferred during recursion. In the case of data links of both special parameters and subgoals, it is a question of how to identify the repeating segment. These method are developed on the foundation of parameter analysis elaborated in Chapter 3. The methods for detecting infinite data transfer sequences will be presented in following algorithms.

Algorithm 4.1 Data Analysis

INPUT : a recursive definition and its set of cyclic parameter links **CPS** (which can be obtained by applying Algorithm 3.3 to the recursive definition. **CPS** is a set of sets {**X**, **ss**, **sp**}. **X** indicates the position of the parameter involved in a cyclic parameter link, **ss** is a set of sets containing subgoals that form the cyclic parameter link, and **sp** is a set of sets containing parameters involved in the cyclic parameter link. (The details of **ss** and **sp** can be found in Algorithm 4.2.1.)

OUTPUT : a message to indicate whether the input recursive definition can terminate

```
NI := 0                                          [ NI: the nontermination indicator ]

If CPS ≠ {} Then
Begin
   For each set in CPS Do                        [ that is, for each cyclic parameter link ]
   Begin
      If only subgoals involved in the cyclic parameter link
         Then Begin
                 apply Algorithm 4.2
                 If the set of cyclic connected data-link lists SCCD ≠ {}
                    Then NI := NI + 1
              End

         Else If only special parameters involved
                 Then Begin
                         apply Algorithm 4.3
                         If the change tendency integer C is not negative
                            Then NI := NI + 1
                         End

                 Else If both subgoals and special parameters involved
                         Then Begin
                                 detect the repeating segment in the data transfer sequence
                                 If there is a repeating segment
                                    Then NI := NI + 1
                                 End

   End
End

If NI = the number of cyclic parameter links
   Then output an appropriate nontermination error message
   Else output a message to indicate that the input recursive definition can terminate
```

Algorithms 4.2 and 4.2.1 are developed for detecting the cyclic connected data-link lists for a cyclic parameter link involving only subgoals.

Algorithm 4.2 Constructing a set of cyclic connected data-link lists for a cyclic parameter link involving only subgoals

INPUT : a set { **X**, **ss**, **sp** } in CPS

OUTPUT : a set of cyclic connected data-link lists **SCCD**, { $(v_1,v_2,...,v_n)$, ... }

SCCD := {}

Apply Algorithm 4.2.1 to { **X**, **ss**, **sp** } to form the set of data links **SDL**, {DL_1, ..., DL_m}, where $DL_i = (v_a, v_b)$

For each data link DL_i, (v_a, v_b), in SDL Do
Begin
  If $v_a = v_b$ Then
  Begin
    SCCD := **SCCD** $\cup$ {DL_i}
    SDL := **SDL** - { DL_i }
  End

SDL_{new} := **SDL**
SDL_{old} := **SDL**

While **i** $\leq$ the number of data links in SDL and $SDL_{new} \neq$ {} Do
[to form the cyclic connected data-link list]
Begin
  SDL_{del} := {}
  SDL_{temp} := {}
  For each DL_i, $(v_a,...,v_b)$, in SDL_{new} Do
    For every data link DL_j, (v_m,v_n) in SDL_{old} Do
      If $v_b = v_m$
        Then If $v_a = v_n$
          Then Begin
            SCCD := **SCCD** $\cup$ { $(v_a,...,v_b,v_n)$ }
            SDL_{del} := SDL_{del} $\cup$ {DL_i, DL_j}
          End
          Else Begin
            SDL_{temp} := SDL_{temp} $\cup$ { $(v_a,...,v_b,v_n)$ }
            SDL_{del} := SDL_{del} $\cup$ {DL_i, DL_j}
          End

Algorithm 4.2 (countined)

$\mathbf{SDL_{new}} := \mathbf{SDL_{new}} - \mathbf{SDL_{del}}$

$\mathbf{SDL_{new}} := \mathbf{SDL_{new}} \cup \mathbf{SDL_{temp}}$

End

eliminate any redundant cyclic connected data-link lists in **SCCD**
[any two cyclic connected data-link lists having the same elements are considered the same, eg., *(1,2,3,1)* and *(2,3,1,2)* are equivalent]

The algorithm below finds data links in a recursive definition for Algorithm 4.2.

Algorithm 4.2.1 Constructing data links for a cyclic parameter link established through only subgoals

INPUT : a set { **X**, **ss**, **sp** } in **CPS**. **ss** = { $\mathbf{s_1}$, $\mathbf{s_2}$, ..., $\mathbf{s_a}$ }. $\mathbf{s_i}$ = { $\mathbf{sgs_1}$, $\mathbf{sgs_2}$, ..., $\mathbf{sgs_b}$ }.
$\mathbf{sgs_i}$ = {$\mathbf{sg_1}$, ..., $\mathbf{sg_c}$}. For a direct recursive definition, **b** = 1.
$\mathbf{sg_i}$ is a subgoal involved in a parameter link.
$\mathbf{sgs_i}$ is a set of all subgoals related to a parameter link for a recursive rule.
$\mathbf{s_i}$ is a set of all $\mathbf{sgs_j}$'s involved in one or more recursive rules in one recursive definition, i.e., a set of sets includes all subgoals for one parameter link.
ss is a set of sets containing all subgoals for one cyclic parameter link.
Similarly, **sp** is a set of sets containing all the parameters involved in a cyclic parameter link corresponding to the subgoals in **ss**.
sp = {$\mathbf{p_1}$, ..., $\mathbf{p_a}$}. $\mathbf{p_i}$ = { $\mathbf{pas_1}$, ..., $\mathbf{pas_b}$ }. $\mathbf{pas_i}$ = { {$\mathbf{X_1}$, ..., $\mathbf{X_{c+1}}$} }.
The number of parameters involved is one more than the number of subgoals involved when one recursive rule is considered. The reason can be shown by considering a simple example. In a recursive rule: *g(A,B) :- link1(A,P1), link2(P1,X), g(X,Y).* There are two subgoals involved: *link1* and *link2*, while there are three parameters *A*, *P1* and *X* involved in the parameter link.

OUTPUT : the set of data links **SDL**, {$\mathbf{DL_1}$, $\mathbf{DL_2}$, ..., $\mathbf{DL_k}$} where each data link is:
$\mathbf{DL_i}$ = ($\mathbf{v_a}$, $\mathbf{v_b}$).

arrange the subgoals in $\mathbf{sgs_i}$ and parameters in $\mathbf{pas_i}$ according to the order of forming the parameter link
[eg., for a recursive rule *a(A,B) :- link1(P2,P1), link2(P1,A), link3(P2,X), a(X,Y).*, the order of the subgoals and parameters forming the parameter link is: *link2, link1, link3* and *A, P1, P2, X* respectively. They are different from their orders appearing in the recursive rule.]

Algorithm 4.2.1 **(countined)**

CV := {} [CV : a set of partial data link values for all parameter links]

For each set $\mathbf{s}_x$ in **ss** Do
Begin [find partial data links for all parameter links]

DV := {} [DV : a set of partial data link values for one parameter link]
[involving all recursive rules in one recursive definition]

For each set $\mathbf{sgs}_y$ in $\mathbf{s}_x$ Do
Begin [find partial data links for all recursive rules]

RV := {} [RV: a set of partial data link values for]
[all subgoals in one recursive rule]

For each subgoal $\mathbf{sg}_z$ in $\mathbf{sgs}_y$ Do
Begin [find partial data links for all subgoals]

V := {} [V: a set of all possible partial]
[data link values for one subgoal]

If the procedure defining subgoal $\mathbf{sg}_z$ does not consist of facts alone
Then transform the original procedure into an equivalent one consisting of facts only

For each fact $\mathbf{sg}_z(..., \mathbf{v}_i, ..., \mathbf{v}_j, ...)$ of $\mathbf{sg}_z$ Do
[$\mathbf{v}_i$ and $\mathbf{v}_j$ are the values corresponding to the parameters $\mathbf{X}_z$ and $\mathbf{X}_{z+1}$ which are used in the
[subgoal $\mathbf{sg}_z(..., \mathbf{v}_i, ..., \mathbf{v}_j, ...)$ to establish the parameter link, they can be found from the set $\mathbf{pas}_y$]
Begin

Find $\mathbf{v}_i$, $\mathbf{v}_j$ from the fact $\mathbf{sg}_z(..., \mathbf{v}_i, ..., \mathbf{v}_j, ...)$, which corresponds to the position of $\mathbf{X}_z$ and $\mathbf{X}_{z+1}$

$\mathbf{V} := \mathbf{V} \cup \{ (\mathbf{v}_i, \mathbf{v}_j) \}$

End

$\mathbf{RV} := \mathbf{RV} \cup \{ \mathbf{V} \}$

End

For each $\mathbf{V}_a$ and $\mathbf{V}_{a+1}$ in **RV** Do
Begin

$\mathbf{V}_{temp} := \{\}$

For each pair $(\mathbf{v}_i, \mathbf{v}_j)$ in $\mathbf{V}_a$ Do

For each pair $(\mathbf{v}_m, \mathbf{v}_n)$ in $\mathbf{V}_{a+1}$ Do

If $\mathbf{v}_j = \mathbf{v}_m$ Then $\mathbf{V}_{temp} := \mathbf{V}_{temp} \cup \{ (\mathbf{v}_i, \mathbf{v}_n) \}$

$\mathbf{V}_{a+1} := \mathbf{V}_{temp}$

End

$\mathbf{DV} := \mathbf{DV} \cup \{ \mathbf{V}_{temp} \}$

End

Algorithm 4.2.1 (countined)

```
   For each V_b and V_b+1 in DV Do
   Begin
      V_temp := {}
      For each pair (v_i, v_j) in V_b Do
         For each pair (v_m, v_n) in V_b+1 Do
            If v_j = v_m Then V_temp := V_temp ∪ { (v_i, v_n) }
      V_b+1 := V_temp
   End

   CV := CV ∪ { V_temp }
End

For each V_c and V_c+1 in CV Do                [find all the data link values]
Begin                                          [for one cyclic parameter link]
   V_temp := {}
   For each pair (v_i, v_j) in V_c Do
      For each pair (v_m, v_n) in V_c+1 Do
         If v_j = v_m Then V_temp := V_temp ∪ { (v_i, v_n) }
   V_c+1 := V_temp
End

SDL := V_temp
```

Algorithm 4.3 is developed for detecting the presence of any infinite data transfer sequence in a recursive definition with cyclic parameter links established through only special parameters.

Algorithm 4.3 detecting the presence of any infinite data transfer sequence for a cyclic parameter link involving only special parameter

INPUT : a set { **X, ss, sp** } in **CPS**

OUTPUT : the *change tendency* of data passing through the cyclic parameter link, measured in terms of an integer **C**; a non-negative value of **C** means that the data transfer sequence is infinite

```
C := 0
```

Algorithm 4.3 (countined)

arrange the parameters in $\mathbf{pas}_i$ according to the order of forming the parameter link

```
CV := {}                          [ CV : a set of change tendency values for all parameter links, ]
                                  [ CV = { C1, ..., Cn }]

For each set px in sp Do
Begin                             [ find the change tendency value for a parameter link ]

   DV := {}                       [ DV : a set of change tendency values for one parameter link ]
                                  [      involving all recursive rules in one recursive definition ]

   For each set pasy in px Do
   Begin                          [ find the change tendency values for all recursive rules ]

      RV := {}                    [ RV: a set of change tendency values for all subgoal parameters ]
                                  [     related to the parameter link in a recursive rule          ]

      For i := 2 to n Do                    [ n = number of parameters in pasy ]
      Begin                                 [ find the change tendency values for all subgoal parameters ]
         If Xi is part of Xi-1 and their lengths
             have a difference of x elements    [ eg., Xi-1 = [H|L] and Xi = L ]
            Then Ctemp1 := - x
            Else Ctemp1 := x
         RV := RV ∪ { Ctemp1 }
      End

      Ctemp2 := 0                           [ Ctemp2 : change tendency values for one recursive rule ]

      For each Ctemp1 in RV Do Ctemp2 := Ctemp2 + Ctemp1

      DV := DV ∪ { Ctemp2 }

   End

   Ctemp3 := 0                              [ Ctemp3 : change tendency values for one parameter link ]

   For each Ctemp2 in DV Do Ctemp3 := Ctemp3 + Ctemp2

   CV := CV ∪ { Ctemp3 }

End

For each Ctemp3 in CV Do C := C + Ctemp3    [ C : change tendency values for ]
                                            [     one cyclic parameter link  ]
```

Since the case of data links formed by mixing both special parameters and subgoals is a rare case, no discussion of how to detect infinite data transfer sequences in this case is provided in this section. As discussed in Section 4.5.2, although an analytical approach for detecting the infinite data transfer sequence in this case is possible, the uninstantiated parameters appearing in data links causes a lot of problems in the detection for the repeating segment of a data transfer sequence. We suggest that a run-time tracing approach may provide a easier approach to detect the repeating segment.

CHAPTER 5—Special Cases

In Chapter 4, we have developed a nontermination detection technique, which we call **data analysis**, based on detecting the presence of any infinite data transfer sequence in any recursive definition of a pure Prolog program. By constructing connected data-link lists, the presence of an infinite data transfer sequence is indicated when any cyclic connected data-link list is found or the size of the data passing through any cyclic parameter link does not tend to become smaller. However, in **the case of interdependent cyclic parameter links**, data analysis may overlook some possible exit conditions and so the cyclic connected data-link list found does not really represent an infinite data transfer sequence. A false warning of nontermination may result in this case. Moreover, in **the cyclic parameter link with special parameters**, data analysis may also be inadequate for detecting nontermination under **a special situation**. We shall see why data analysis may fail in the two special cases and suggest necessary modifications on data analysis to remedy the problem. Since all the special cases are rare cases in Prolog programming, we shall only discuss the general concept underlying the necessary modifications and leave the unnecessary details behind. Therefore, no algorithms will be given in this chapter.

5.1 Interdependent Cyclic Parameter Links

As mentioned in the beginning of Chapter 4, it is possible to form **interdependent** cyclic parameter links through **some common parameters** and/or **common subgoals if** there exist more than one cyclic parameter link in a recursive **definition. Since data** analysis developed in Chapter 4 is expected to handle recursive **definitions with only** independent cyclic parameter links, **the technique may yield an erroneous conclusion**

when it is applied to a recursive definition with some interdependent cyclic parameter links. For instance, data analysis may incorrectly lead to the conclusion that nontermination can occur in a certain recursive definition when its interdependent cyclic parameter links can work together to provide an exit condition at a certain point of the recursion. In this section, we shall first discuss how interdependent and independent cyclic parameter links are different from each other. The discussion can reveal to us how the presence of interdependent cyclic parameter links can provide a unique way to avoid nontermination. Then **we shall explore how data analysis can be adapted to a recursive definition consisting of some interdependent cyclic parameter links.**

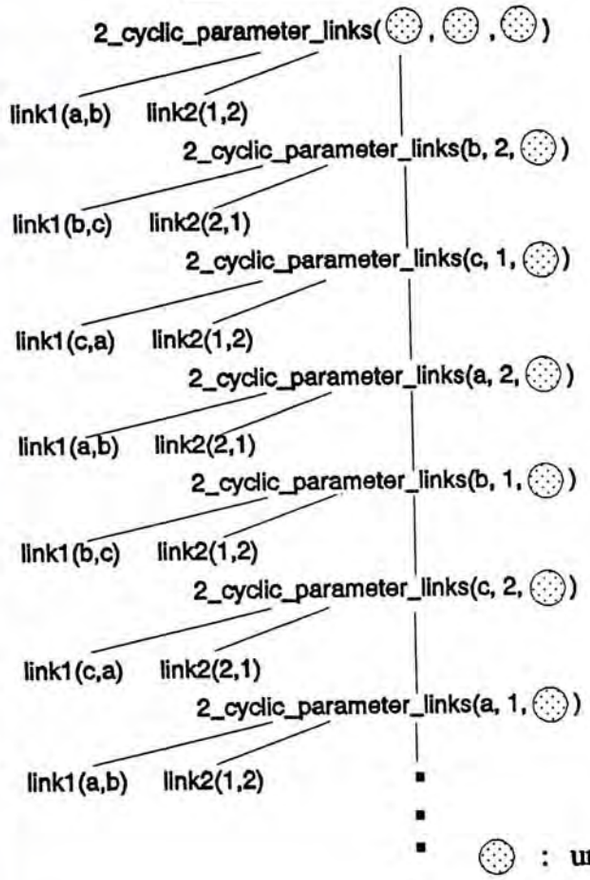
5.1.1 Interdependent Cyclic Parameter Links through Common Parameters

Program (b) in Figure 5.1 provides an example of **interdependent cyclic parameter links sharing a common parameter**, while Program (a) consists of independent cyclic parameter links only. By contrasting Program (b) with Program (a), we can see how interdependent and independent cyclic parameter links are differentially related to the exit-reaching process. When Program (a) is examined, we can find that it is almost identical to Program (b) except that its two cyclic parameter links are formed by one subgoal instead of two. However, nontermination can occur in Program (a) but not Program (b). In Program (a), two independent cyclic parameter links are formed between the parameters *P1in* and *P1out* through the subgoal *link1*, and between the parameters *P2in* and *P2out* through the subgoal *link2*. Each of them has data links that can form a cyclic connected data-link list. Hence, each of them has an infinite data transfer sequence during the evaluation of the recursive definition so that none of them can perform as an exit-reaching process. The evaluation of the recursive definition will result in nontermination. This is clearly shown by Search tree (a) in Figure 5.1. However, nontermination can be eliminated by a minor modification on these two cyclic parameter links. Program (b) shows how the modification can be made.

Program (a) the recursive definition with two independent cyclic parameter links that leads to nontermination

```
2_cyclic_parameter_links(P1in, P2in, X) :-
        link1(P1in, P1out), link2(P2in, P2out),
          2_cyclic_parameter_links(P1out, P2out, X).
2cyclic_parameter_links(a, 1, not_end).

        link1(a, b).   link2a(1, 2).
        link1(b, c).   link2a(2, 1).
        link1(c, a).
```

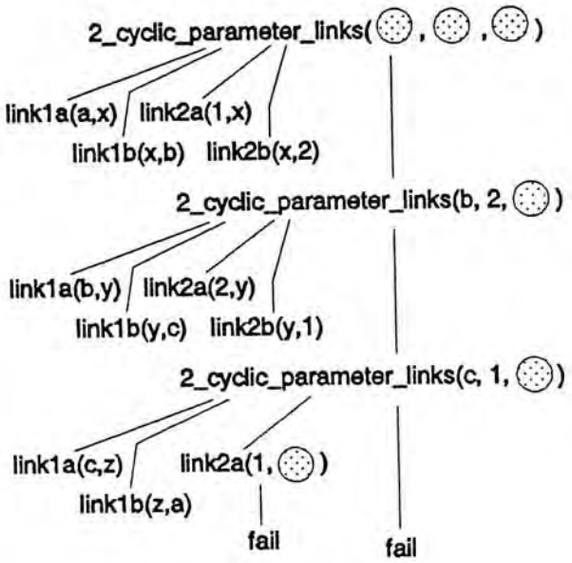
Search Tree (a)



Program (b) the recursive definition with two dependent cyclic parameter links that can terminat

```
2_cyclic_parameter_links(P1in, P2in, X) :-
        link1a(P1in, CP), link1b(CP, P1out),
          link2a(P2in, CP), link2b(CP, P2out),
            2_cyclic_parameter_links(P1out, P2out, X).
2cyclic_parameter_links(a, 1, not_end).

link1a(a, x).  link1b(x, b).    link2a(1, x).  link2b(x, 2).
link1a(b, y).  link1b(y, c).    link2a(2, y).  link2b(y, 1).
link1a(c, z).  link1b(z, a).
```

Search Tree (b)



: uninstantiated parameter

Figure 5.1

In Program (b), each cyclic parameter link is established out of two subgoals instead of one. By using two subgoals, we can introduce another parameter into each cyclic parameter link. As the subgoal *link1(P1in, P1out)* in Program (a) is now broken down into two subgoals, *link1a(P1in, CP)* and *link1b(CP, P1out)*, the new parameter *CP* is introduced to the cyclic parameter link between parameters ***P1in*** and ***P1out***. In a similar way, the parameter *CP* is also introduced to the cyclic parameter link between

P2in and *P2out*. Through the common parameter *CP*, the two cyclic parameter links have now become interdependent because infinite data transfer sequences can exist in these two cyclic parameter links only if both data transfer sequences have the same value to pass through the common parameter in every level of recursion. For example, the second argument of both the subgoals *link1a* and *link2a* constitutes the common parameter *CP*. In order to allow data to pass through them, the same value must be instantiated to both of their second arguments. Therefore, when the fact *link1a(b,y)* is instantiated to the subgoal *link1a*, the subgoal *link2a* must be instantiated with the fact *link2a(2,y)* to allow the same value *y* to be instantiated to the common parameter *CP* in both subgoals. Therefore, the data transfer in one cyclic parameter link and the data transfer in another cyclic parameter link affect each other due to the presence of a common parameter and that is the reason why the two cyclic parameter links are considered to be interdependent.

If each of the cyclic parameter links in Program (b) is examined separately, it has data links that can form a cyclic connected data-link list: *a--b--c--a* for the first cyclic parameter link and *1--2--1* for the second one. So data analysis will conclude that nontermination can arise during the evaluation of the recursive definition because all of its cyclic parameter links cannot function as an exit-reaching process. However, Search tree (b) clearly shows that such a conclusion is erroneous. The evaluation of the recursive definition in Program (b) does terminate at the third level of recursion because the subgoal *link2a* fails and blocks further backtracking. If we examine why the subgoal *link2a* fails at this point, we can see that it fails because the only value that can be instantiated to the common parameter *CP* in the cyclic parameter link formed by the subgoals *link1a* and *link1b* cannot agree with the only value that can be instantiated to the same parameter *CP* in the cyclic parameter link formed by the subgoals *link2a* and *link2b*. Since the procedure of the subgoal *link2a* only consists of the facts *link2a(1,x)* and *link2a(2,y)*, the parameter *CP*, as the second argument of the subgoal *link2a*, can be instantiated only to either *x* or *y* in the second cyclic parameter link. But the same parameter *CP* must be instantiated to be *z* in the first cyclic parameter link formed by the subgoals *link1a* and *link1b* when the recursion reaches the third level. The conflict of values at the common parameter shared between two interdependent cyclic parameter

links blocks any further recursion. This can be easily perceived if we compare the recursive definition in Program (b) with a modified one as follows:

```
2_cyclic_parameter_links(P1in, P2in, X) :-
        link1a(P1in, CP1),  link1b(CP1, P1out),
                link2a(P2in, CP2),  link2b(CP2, P2out),
                        2_cyclic_parameter_links(P1out, P2out, X).
2_cyclic_parameter_links(a, 1, not_end).

link1a(a,x).        link1b(x,b).        link2a(1,x).        link2b(x,2).
link1a(b,y).        link1b(y,c).        link2a(2,y).        link2b(y,1).
link1a(c,z).        link1b(z,a).
```

By only replacing the common parameter *CP* in both cyclic parameter links with two different parameters *CP1* and *CP2*, we can eliminate the blockage completely. The values instantiated to the parameter *CP1* no longer affect the values instantiated to the parameter *CP2*. Therefore, the two cyclic parameter links become independent again. When this modified version of Program (b) is examined, we can see that it actually behaves like as Program (a). By sharing a common parameter, some restrictions are imposed on the values that can be transferred through the common parameter of these cyclic parameter links. By removing the common parameter, the restrictions can be lifted. Using a data transfer analogy, we can view the common parameter as a common channel for two (or more) data transfer sequences. Although each data transfer sequence is infinite if they are considered separately, the blockage at common channel can block all the involved data transfer sequences. Due to this characteristic of the common parameter in interdependent cyclic parameter links, the common parameter can act as an exit condition in some situations. Consequently, the cyclic connected data-link list detected by data analysis cannot be a proper indicator of an infinite data transfer sequence in a recursive definition with some interdependent cyclic parameter links if no consideration is paid to the values passing through the common parameter. Therefore, the method developed in Chapter 4 can yield invalid conclusions in the case of interdependent cyclic parameter links in some situations.

However, **the interdependent cyclic parameter links do not necessarily form an exit condition.** The blockage at common parameter can be avoided without eliminating the common parameter. By comparing the recursive definition in Program (b) in Figure 5.1 to the one in Figure 5.2, we can see that there is no difference between these two recursive definitions except for the procedures of the subgoals *link2a* and *link2b*, in which the common parameter *CP* is involved. If we examine these two procedures, we can find how they are carefully rewritten to avoid the conflict in the value passing through the

the recursive definition modified from Program (b) in Figure 5.1 that also leads to nontermination despite the presence of two dependent cyclic parameter links

```
2_cyclic_parameter_links(P1in, P2in, X) :-
        link1a(P1in, CP), link1b(CP, P1out),
          link2a(P2in, CP), link2b(CP, P2out),
                2_cyclic_parameter_links(P1out, P2out, X).
2cyclic_parameter_links(a, 1, not_end).

link1a(a, x). link1b(x, b).     link2a(1, x). link2b(x, 2).
link1a(b, y). link1b(y, c).     link2a(2, y). link2b(y, 1).
link1a(c, z). link1b(z, a).     link2a(1, z). link2b(z, 2).
                                link2a(2, x). link2b(x, 1).
                                link2a(1, y). link2b(y, 2).
                                link2a(2, z). link2b(z, 1).
```

Search Tree for the above recursive definition

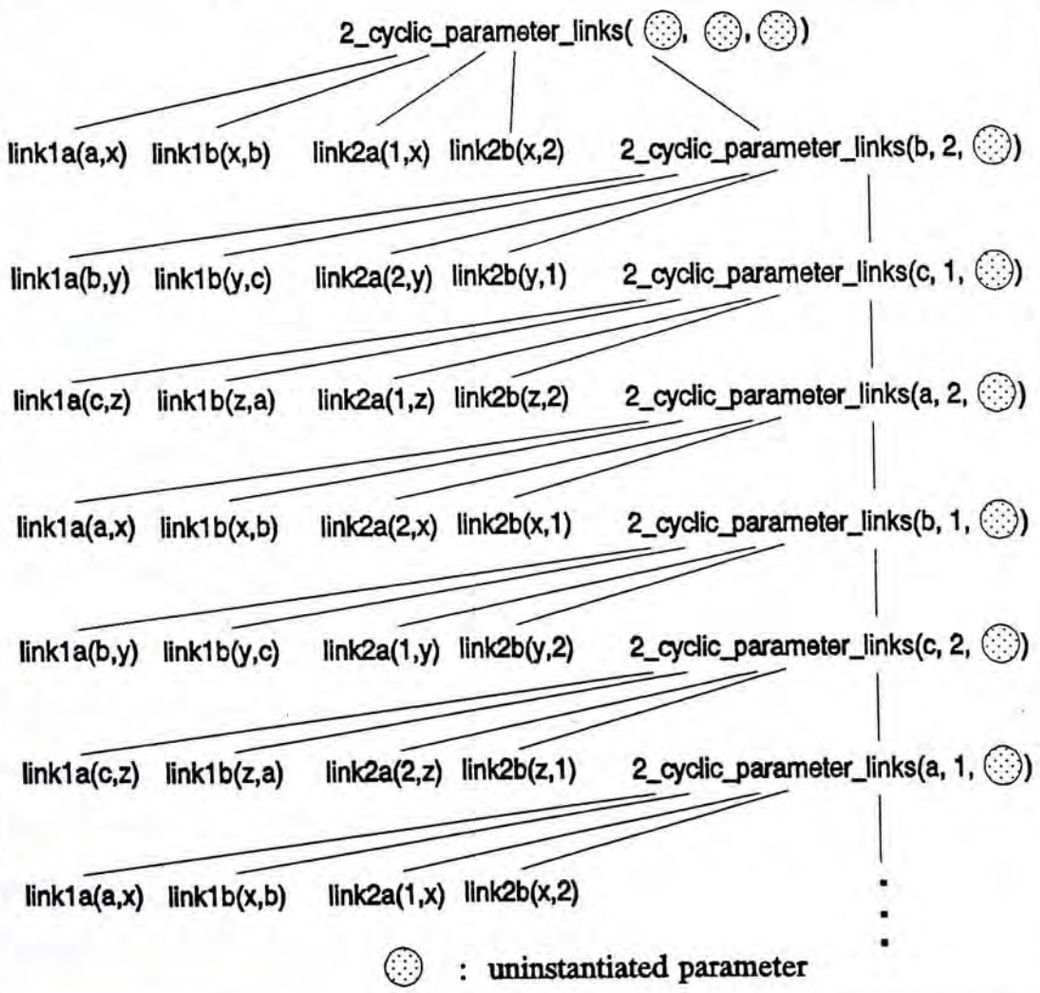


Figure 5.2

common parameter. By expanding the procedure of the subgoals *link2a* and *link2b*, the conflict in the value passing through the common parameter can be solved. The search tree in Figure 5.2 shows how the newly added facts in the procedures of the subgoals *link2a* and *link2b* can allow the common parameter *CP* to be instantiated to the same value in different levels of recursion. Although the size of the procedures of the subgoals *link2a* and *link2b* in Figure 5.2 is double the size of the procedures of *link2a* and *link2b* in Program (b) in Figure 5.1, only one cyclic connected data-link list *1--2--1* exists in both cases. **The newly added facts** do not increase the number of cyclic connected data-link list in any cyclic parameter link but **form more alternative paths to allow data to be transferred through the common parameter.** The example in Figure 5.2 also shows that data analysis developed in Chapter 4 is not completely irrelevant in the case of interdependent cyclic parameter links. Since at least one cyclic connected data-link list is present in all the cyclic parameter links, if data analysis in Chapter 4 is applied to the recursive definition in Figure 5.2, a correct conclusion will be drawn: nontermination can occur, just as what is indicated by the search tree in Figure 5.2.

Therefore, the examples in Figures 5.1 and 5.2 show that, in the case of interdependent cyclic parameter links, **the mere presence of the cyclic connected data-link lists** in those interdependent cyclic parameter links **is not sufficient** to indicate the presence of any infinite data transfer sequence. On the other hand, **the mere presence of interdependent cyclic parameter links cannot constitute an exit condition by itself.** Infinite data transfer sequences can be formed only if the cyclic connected data-link lists found in the interdependent cyclic parameter links do not form any blockage at common parameter shared by these interdependent cyclic parameter links. In other words, the values passing through the common parameter in all the involved interdependent cyclic parameter links must be the same. Therefore, the presence of interdependent cyclic parameter links does not require a completely different method. Instead, after cyclic connected data-link lists are detected by data analysis, we only need **a certain method to verify whether these cyclic connected data-link lists can form any blockage at common parameter.**

Case (I)	Case (II)
the common parameter is also the parameter of the recursive rule head and the recursive subgoal	the common parameter is only the parameter of some subgoal in the recursive rule
goal(AX,BX,CX) :- link1a(AX,XB), link1b(BX,XA), link2a(BX,XC), link2b(CX,XB), goal(XA,XB,XC).	goal(AX,BX,CX,DX) :- link1a(XA,XB), link1b1(BX,CP), link1b2(CP,XA), link2a(CX,XD), link2b1(DX,CP), link2b2(CP,XC), goal(XA,XB,XC,XD).

Figure 5.3

However, **there are two ways to form interdependent cyclic parameter links through common parameters.** The difference between them can be illustrated by the two recursive rules in Figure 5.3. We can find that two interdependent cyclic parameters in Case (II) are formed in the same way as other examples shown in Figures 5.1 and 5.2 through the common parameter *CP*. However, the interdependent cyclic parameter links formed in this way can be easily converted into independent ones. The subgoals *link1b1(BX,CP)* and *link1b2(CP,XA)* can be simply merged into one subgoal *link1b(BX,XA)* to eliminate the common parameter *CP* without any significant effect on the other cyclic parameter link. It is also true for the subgoals *link2b1* and *link2b2*. But the common parameters in the interdependent cyclic parameter links in Case (I) cannot be eliminated in this manner. In Case (I), all the common parameters *BX* and *XB* are also the parameters in the recursive rule head or the recursive subgoal. They cannot be removed without significantly altering the recursive definition itself. Therefore, before we discuss how to revise data analysis, we must find out whether the interdependent cyclic parameter links formed in these two different manners are significantly different in terms of nontermination detection. By comparing the interdependent cyclic parameter links in Case (I) with those in Case (II), we can find that the most significant difference is this: while there are definitely two different cyclic parameter links in Case (II), **the two interdependent cyclic parameter links in Case (I) can be considered as one longer cyclic parameter link** extended over four levels of recursion. If cyclic connected

data-link lists are found in the two interdependent cyclic parameter links and no conflict of values occurs at the common parameter between these cyclic connected data-link lists, we can find that at least one cyclic connected data-link list can be formed in the longer cyclic parameter link formed by combining the two shorter interdependent cyclic parameter links. However, if a cyclic connected data-link list can be formed among the subgoals forming the longer cyclic parameter link, this indicates that an infinite data transfer sequence can be formed without blockage at common parameter and thus nontermination will occur. The same reasoning that is used to develop the method of data analysis in Chapter 4 can also be applied in this case. In other words, the test for nontermination described in Chapter 4 can also be applied to the recursive definitions of Case (I) without any modification with the only constraint that it must be **conducted on the longer cyclic parameter link instead of the shorter interdependent cyclic parameter links.** But the two interdependent cyclic parameter links **in Case (II) cannot form a longer cyclic parameter link.** We need to develop a certain technique to check whether the blockage of data transfer can happen in every common parameter shared among the interdependent cyclic parameter links. Therefore, it is Case (II) that requires modifications to the method of data analysis developed in Chapter 4. Since the problem of the presence of common parameters in both cases is the problem of how to know whether blockage of data transfer can occur at the common parameters, any adaptation of the method of data analysis to handle the presence of interdependent cyclic parameter links in Case (II) can also be applied to Case (I). Our discussion therefore will concentrate on the interdependent cyclic parameter link formed by the type of common parameters shown in Case (II).

5.1.1.1 Interdependency between Cyclic and Non-cyclic Parameter Links and Interdependency between Cyclic Parameter Link and Subgoals

Before examining interdependent cyclic parameter links, we shall first consider the interdependency between a cyclic parameter link and something other than a cyclic parameter link. That is, through a common parameter, a non-cyclic parameter link or a subgoal can also establish interdependency with a cyclic parameter link. Similar to the

case of interdependent cyclic parameter links, data analysis may sometimes yield erroneous conclusions about nontermination when a cyclic parameter link shares a common parameter with some non-cyclic parameter links or subgoals. An appreciation of this case can help us to understand the case of interdependent cyclic parameter links.

In the discussion on the modified version of the recursive definition in Figure 4.3 (in Section 4.1.1.1), we have already shown an example in which the subgoals that are not part of a cyclic parameter link can significantly affect the construction of the data link in the corresponding cyclic parameter link. If we examine the recursive definition in Figure 4.3 again, we can find that the common parameter *X* shared between the cyclic parameter link and the subgoal *link2* is crucial. Comparing the case in Figure 4.3 and its modified version discussed in Section 4.1.1.1 with the interdependent cyclic parameter links in Figures 5.1 and Figure 5.2, we can find that the common parameter plays the same role in both cases. Due to the common parameter, the values transferred through the common parameter need to be regulated by all the subgoals sharing this common parameter, just like the case of interdependent cyclic parameter links, in which all the cyclic parameter links sharing the same parameter can regulate the values passing through it.

By the same reason, a cyclic parameter link and a non-cyclic parameter link are considered to be interdependent if both are sharing at least one common parameter. In Figure 5.4, we form a recursive definition that has interdependent cyclic and non-cyclic parameter links by slightly modifying the recursive definition in Figure 4.3. By changing the subgoal *link2(Z,X)* to *link2(Q,X)*, the subgoal *link2* form a non-cyclic parameter link between the first and second parameter of the recursive definition. (It can show how closely a non-cyclic parameter link and a mere subgoal are related to each other.) The procedures of the subgoals used in the recursive rule are also modified so that two cyclic connected data-link lists can be present in the cyclic parameter link formed by the subgoals *link1*, *link3* and *link4*. Since it is the only cyclic parameter link in the recursive definition, data analysis will indicate that the recursive definition is nonterminating if the cyclic parameter link is considered in isolation. However, the cyclic parameter link is interdependent with the non-cyclic parameter link formed by the subgoal *link2*. In

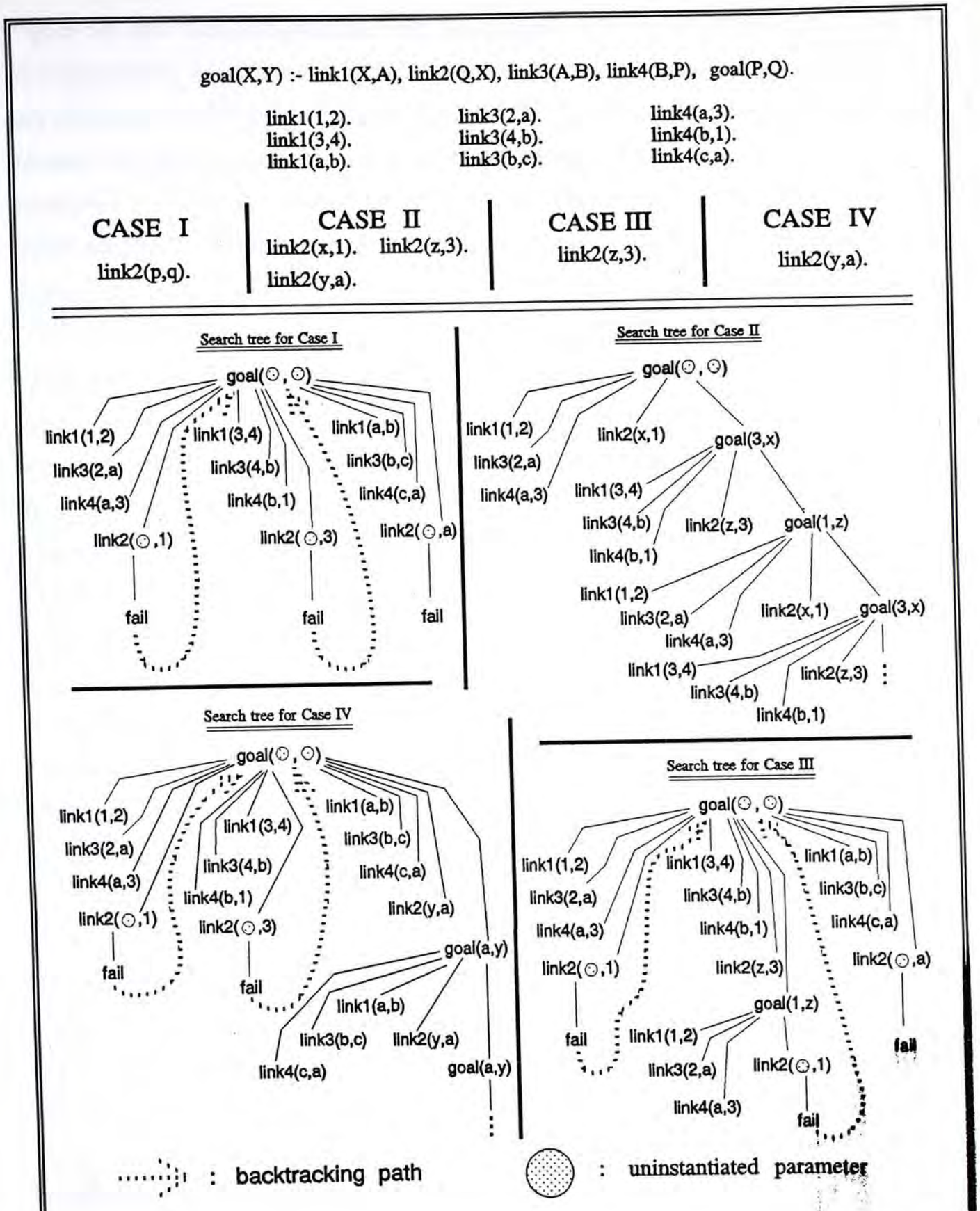

goal(X,Y) :- link1(X,A), link2(Q,X), link3(A,B), link4(B,P), goal(P,Q).
link1(1,2).
link1(3,4).
link1(a,b).
link3(2,a).
link3(4,b).
link3(b,c).
link4(a,3).
link4(b,1).
link4(c,a).
CASE I
link2(p,q).
CASE II
link2(x,1). link2(z,3).
link2(y,a).
CASE III
link2(z,3).
CASE IV
link2(y,a).
Search tree for Case I
goal(⊙,⊙)
link1(1,2)
link3(2,a)
link4(a,3)
link2(⊙,1)
fail
link1(3,4)
link3(4,b)
link4(b,1)
link2(⊙,3)
fail
link1(a,b)
link3(b,c)
link4(c,a)
link2(⊙,a)
fail
Search tree for Case II
goal(⊙,⊙)
link1(1,2)
link3(2,a)
link4(a,3)
link2(x,1)
goal(3,x)
link1(3,4)
link3(4,b)
link4(b,1)
link2(z,3)
goal(1,z)
link1(1,2)
link3(2,a)
link4(a,3)
link2(x,1)
goal(3,x)
link1(3,4)
link3(4,b)
link4(b,1)
link2(z,3)
Search tree for Case IV
goal(⊙,⊙)
link1(1,2)
link3(2,a)
link4(a,3)
link2(⊙,1)
fail
link1(3,4)
link3(4,b)
link4(b,1)
link2(⊙,3)
fail
link1(a,b)
link3(b,c)
link4(c,a)
link2(y,a)
goal(a,y)
link1(a,b)
link3(b,c)
link4(c,a)
link2(y,a)
goal(a,y)
Search tree for Case III
goal(⊙,⊙)
link1(1,2)
link3(2,a)
link4(a,3)
link2(⊙,1)
fail
link1(3,4)
link3(4,b)
link4(b,1)
link2(z,3)
goal(1,z)
link1(1,2)
link3(2,a)
link4(a,3)
link2(⊙,1)
fail
link1(a,b)
link3(b,c)
link4(c,a)
link2(⊙,a)
fail
: backtracking path
: uninstantiated parameter

Figure 5.4

Figure 5.4, four different procedures for the subgoal *link2* are given to illustrate how the interdependency between the cyclic parameter link and the non-cyclic parameter link can eliminate nontermination in some cases. In Case I, nontermination is eliminated because the fact defining the subgoals *link2*, *link2(p,q)* does not allow the common parameter *X* to share the same value in both parameter links at the same time. The search tree for Case I in Figure 5.4 shows this clearly. In order to have data transferring through the cyclic parameter link, the parameter *X* must be instantiated to the value of *1*, *3* or *a* in the cyclic parameter link. But the procedure defining the subgoal *link2* which forms the interdependent non-cyclic parameter link only allow the parameter *X* to be instantiated to the value *q*. The conflict of values blocks the infinite data transfer sequence in the cyclic parameter link so that nontermination does not result. Similarly, the procedure of *link2* in Case III produces the same effect on the infinite data transfer sequence in the interdependent cyclic parameter link. As indicated by the process to form the cyclic connected data-link list *1--3--1*, both facts *link1(1,2)* and *link1(3,4)* are essential. Hence, the infinite data transfer sequence corresponding to the connected data-link list *1--3--1* can exist only if both the values *1* and *3* can pass through the common parameter *X*. Although the fact *link2(Z,3)* can allow the value *3* to be shared between two interdependent parameter links, the procedure defining the subgoal *link2* cannot allow the value *1* to pass through the common parameter. It can be shown by the search tree for Case III. The infinite data transfer sequence thus cannot be completed and no nontermination can happen. On the other hand, the presence of the additional fact *link2(x,1)* in Case II makes it possible to transfer the value *1* through the common parameter. So, the infinite data transfer sequence is now completed and nontermination occurs, as indicated by its corresponding search tree. Nontermination also happens in Case IV. The search tree for Case IV shows that it is caused by a similar reason. With the fact *link2(y,a)* in its procedure, the non-cyclic parameter link allows only the value *a* to pass through the common parameter *X*. However, a cyclic connected data-link list, *a--a*, can be established in the cyclic parameter link if the value *a* can be transferred through the common parameter. Because only one infinite data transfer sequence is sufficient to cause nontermination to arise, nontermination occurs in Case IV as well as in Case II. In fact, by examining the search trees for Case II and Case IV, we can find that only one infinite data transfer sequence is followed during the

recursion. Even though the fact *link2(y,a)* is also present in the procedure of *link2* in Case II, only the infinite data transfer sequence corresponding to the cyclic connected data-link list *1--3--1* is followed in Case II. By comparing these cases and their corresponding search trees, we can conclude that nontermination can occur if

(1) there is at least a cyclic connected data-link lists in the interdependent cyclic parameter link; and

(2) the value(s) passing through the common parameter according to the cyclic connected data-link list in the interdependent cyclic parameter link can agree with the value(s) passing through the same common parameter according to any data transfer sequence of the interdependent non-cyclic parameter link.

If a subgoal that is not part of any parameter link happens to share a common parameter with a subgoal sharing some common parameter with the subgoal in a cyclic parameter link, the effect is the same as in the case of interdependent cyclic and non-cyclic parameter links. Consequently, the method of data analysis can be adapted to the case of cyclic parameter links sharing a common parameter with other subgoals or non-cyclic parameter links. The following are the steps in the adapted method:

(1) Find out the common parameter(s) shared between the interdependent cyclic parameter link and the interdependent non-cyclic parameter link or subgoal.

(2) If any cyclic connected data-link list can be found in an interdependent cyclic parameter link, for each data link used to construct a cyclic connected data-link list, find out the value(s) transferred through the common parameter(s). (The value(s) can be found either by simply checking up the facts in the procedure(s) defining the subgoal(s) with the common parameter(s) involved, or by analyzing the rule(s) in the corresponding procedure(s) in a way similar to how connected data-link list is constructed in data analysis.) As suggested in Figure 5.4, the common parameter value(s) can be inserted into the cyclic connected data-link list as part of it.

(3) Apply step (2) to the interdependent non-cyclic parameter link or subgoal to construct in them all the possible connected data-link lists with the value(s) passing through the common parameter indicated.

(4) For each cyclic connected data-link list found in step (2), compare the value(s) passing the common parameter to the value(s) passing through the same common parameter found in step (3).

(5) If at least one cyclic connected data-link list of the interdependent cyclic parameter link can match all the value(s) of its common parameter(s) to any common parameter value(s) in any connected data-link list of the interdependent non-cyclic parameter link or subgoal, nontermination occurs during the evaluation of the corresponding recursive definition; otherwise, the evaluation of the recursive definition can terminate.

If we apply the above steps to Case III, the parameter *X* can be identified as the common parameter in step (1). In the only cyclic parameter link, which is formed by the subgoals *link1*, *link3* and *link4*, there are two cyclic connected data-link lists in the cyclic parameter link: *1--3--1* and *a--a*. In step (2), the facts constructing the two cyclic connected data-link lists are examined to find out what values can be transferred through the common parameter: the values for the cyclic connected data-link list *1--3--1* are *1* and *3*, while the value for the cyclic connected data-link list *a--a* is *a*. Then the connected data-link list of the non-cyclic parameter link is constructed in step (3) and the value passing through the common parameter is recorded. Only the value *3* is possible. By comparing the value found in step (3), we can see that *3* does not match *a*. Therefore, the cyclic connected data-link list *a--a* can be eliminated from our consideration. On the other hand, although the value from step (3) can match one of the values of the common parameter in the cyclic connected data-link list *1--3--1*, not all the values of the common parameter in this cyclic connected data-link list can be matched. Hence, nontermination does not occur.

On the other hand, if Case IV is considered, even though the application of steps (1) and (2) yields the same result as in Case III: the values of the common parameter in the cyclic connected data-link list *1--3--1* are *1* and *3* and the value of the common parameter in the set *a--a* is *a*, the value passing through the common parameter for the non-cyclic parameter link is found to be *a* in step (3). As these values are compared in step (4), all the values of the common parameter in the cyclic connected data-link list *a--a* can be matched to the values of the common parameter for the non-cyclic parameter link found in step (3). Therefore, we can conclude that nontermination will result during the evaluation of the recursive definition in Figure 5.4 with the procedure

of the subgoal *link2* in Case IV. In other words, nontermination occurs only if at least an infinite data transfer sequence can be formed in the interdependent cyclic parameter links without blockage at common parameter shared by other subgoals in the recursive definition. The above steps therefore can be added to data analysis to detect whether any blockage exists at the point where the potential infinite data transfer sequence passes through each common parameter. These steps can eliminate the possibility of false warning.

Although the examples shown in Figure 5.4 consist of only non-cyclic parameter links of one recursion level, the above steps are also applicable to non-cyclic parameter links extending over **multiple levels of recursion**. Step (3) is not limited to one-level non-cyclic parameter links as long as the interdependent cyclic and non-cyclic parameter links share only one common parameter. If they both share only one common parameter, the other part of the multi-level non-cyclic parameter link cannot affect the other common parameters on the interdependent cyclic parameter link. For **interdependent cyclic and non-cyclic parameter links that share more than one common parameter**, the above method requires some modification. In some situations, blockage at common parameter can be avoided only if **all the common parameters** shared between the interdependent cyclic and non-cyclic parameter links come from the **same cyclic connected data-link list** formed from the cyclic parameter links and the **same connected data-link list** formed from the non-cyclic parameter links. Although the steps (2) and (4) can make sure that only common parameter from the same cyclic connected data-link list will be tested, the steps (4) and (5) will allow the values of common parameters from different connected data-link lists from the non-cyclic parameter link to match to the values of the common parameters in the interdependent cyclic parameter link. Therefore, the above method may yield an erroneous conclusion in some situations. However, since the interdependent cyclic and non-cyclic parameter links with more than one common parameter is a very rare case in Prolog programming, (it is extremely confusing even to the programmer himself/herself,) we shall not go into detail in this thesis.

5.1.1.2 Interdependency between Cyclic Parameter Links

Similar to the case in which a common parameter is shared by non-cyclic parameter links and cyclic parameter links, two or more cyclic parameter links become interdependent cyclic parameter links when they share a common parameter. In Figure 5.1 and Figure 5.2, we have already seen some examples of interdependent cyclic parameter links. As shown in these examples, sharing a common parameter in these cyclic parameter links can provide a way to form an exit condition, which is not present in the case of independent cyclic parameter links. Since data analysis has not considered such a possibility, some modification to the method is needed. However, unlike the case of interdependent cyclic and non-cyclic parameter links described in the previous section, the cyclic connected data-link lists in these interdependent cyclic parameter links usually involve more than one common parameter.

In Figure 5.5, there is a recursive definition with two interdependent cyclic parameter links. The first cyclic parameter link is located between parameters *AX/XA* and *BX/XB*, while the second one is located between parameters *CX/XC* and *DX/XD*. By analyzing the subgoals in the recursive rule, we can see that the two cyclic parameter links are of the same length. Both extend over two levels of recursion. Between the interdependent cyclic parameter links, only one common parameter *CP* is shared. On the other hand, the cyclic connected data-link lists of these two cyclic parameter links extend over two cycles of its corresponding cyclic parameter link. In other words, the cyclic connected data-link lists from both interdependent cyclic parameter links extend over four levels of recursion. Since they **both extend over two cycles of the cyclic parameter links** that share a common parameter, the two cyclic connected data-link lists **share two common parameters.** It can be shown by the graphical representation in Figure 5.5. We can trace one complete cycle of the two cyclic connected data-link lists from both cyclic parameter links to see how the two common parameters are shared between the two cyclic connected data-link lists.

goal(AX,BX,CX,DX) :- link1a1(AX,CP), link1a2(CP,XB), link1b(BX,XA),
link2a(CX,XD), link2b1(DX,CP), link2b2(CP,XC), goal(XA, XB, XC, XD).

link1a1(1,a).	link1a1(3,b).	link2a(11,12).	link2a(13,14).
link1a2(a,2).	link1a2(b,4).	link2b1(12,a).	link2a1(14,b).
link1b(2,3).	link1b(4,1).	link2b2(a,13).	link2b2(b,11).

the graphical representation showing values transferring through each parameter of all the cyclic parameter links in the above recursive definition

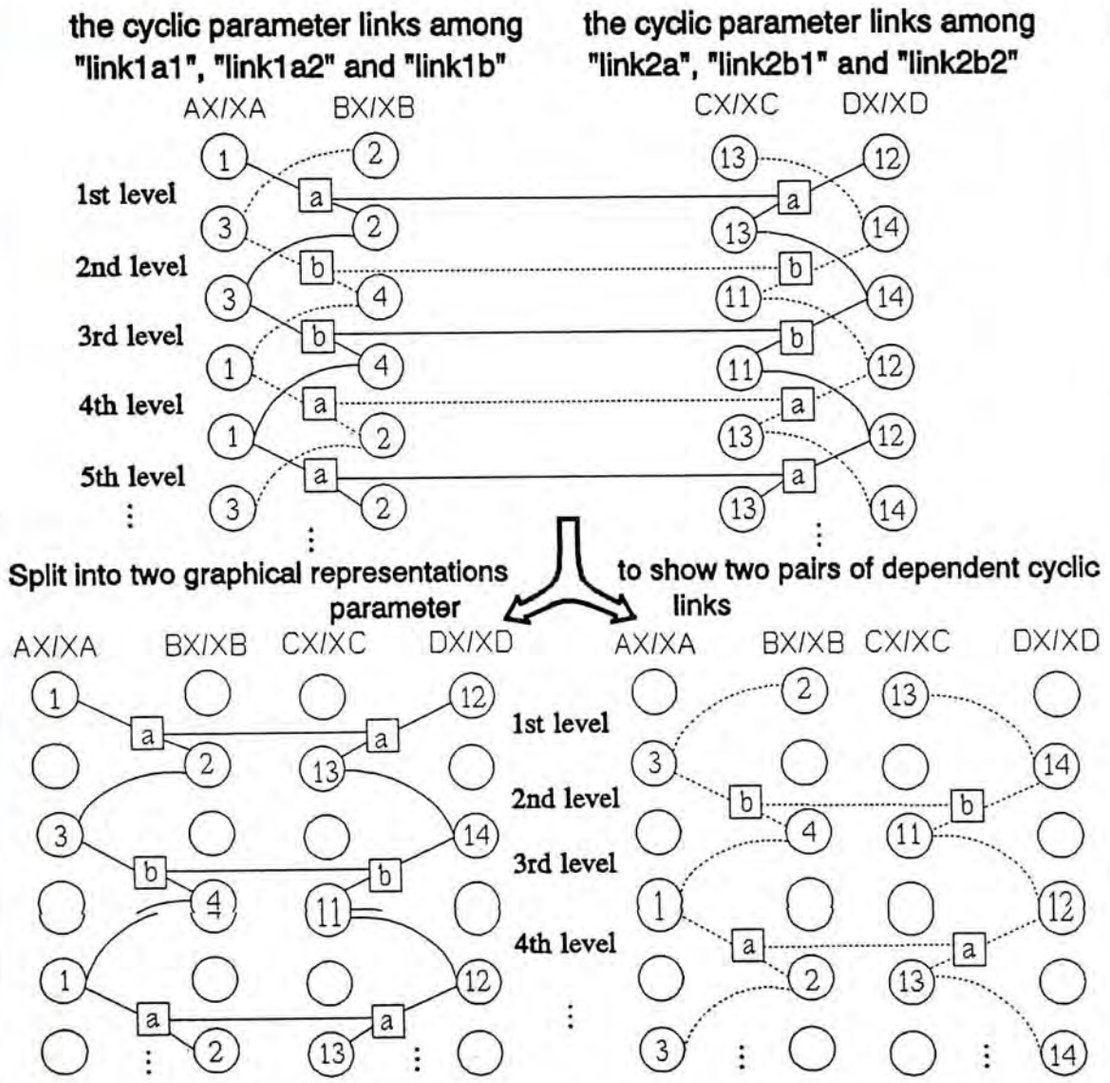


Figure 5.5

In Figure 5.5, we modify the graphical representation scheme developed in Chapter 3 to illustrate how the common parameters are shared between two interdependent cyclic parameter links. Circles indicate the parameters responsible for transferring data from one level of recursion to next level, i.e., the parameters *AX*, *BX*,

CX, *DX*, *XA*, *XB*, *XC* and *XD*, which are the parameters in the recursive rule head or in the recursive subgoal. **Squares** indicate **the common parameters** shared between the two interdependent cyclic parameter links. The squares are located between two circles because they are responsible for the formation of a parameter link from the parameter in the recursive rule head to the parameter in the recursive subgoal. If the two interdependent cyclic parameter links can transfer the same value through the common parameter at a particular level of recursion, we indicate **the absence of blockage at common parameter** by **the squares connected with a solid or dotted line.**

The modified graphical representation also shows that **more than one cyclic parameter link are formed among the same set of parameter.** For the parameters *AX/XA* and *BX/XB*, there are actually two cyclic parameter links. When we re-examine the recursive definition, we can see that all subgoals needed to form the same cyclic connected data-link list exist in the same recursive rule. Hence, in each level of recursion, two parameter links exist between the parameters *AX/XA* and *BX/XB*. Eventually, these parameter links are connected to form two cyclic parameter links if the successive levels of recursion are considered. Therefore, as shown by the graphical representation in the center of Figure 5.5, there are actually two pairs of interdependent cyclic parameter links instead of one pair. These two pairs of interdependent cyclic parameter links are represented by the dotted line and solid line respectively. If only one pair of interdependent cyclic parameter links is considered, it seems to be that no value is transferred through the common parameter in every alternate level of recursion. However, the evaluation of the recursive definition can continue only if some value can be passed through the common parameter *CP* in every level of recursion; otherwise, the subgoals with *CP* as one of its argument, *link1a1*, *link1a2 link2b1* and *link2b2*, will fail and block further recursion. The graphical representation in the center of Figure 5.5 shows us how two pairs of interdependent cyclic parameter links can work together to pass values through the common parameter in every level of recursion.

However, it is not necessary to examine all cyclic connected data-link lists in all pairs of interdependent cyclic parameter links. The example in Figure 5.5 shows us that, in some situation, examining only the cyclic connected data-link lists from one of the

several pairs of interdependent cyclic parameter links is sufficient to know whether blockage at common parameter will occur. At the bottom of Figure 5.5, we split the original graphical representation into two, in which only one pair of interdependent cyclic parameter links is shown. They clearly show that **the pairs of cyclic connected data-link lists used in both pairs of interdependent cyclic parameter link are actually the same.** The graphical representation in the bottom right hand corner is identical to the result obtained by shifting the one at the bottom left hand corner one level up. The reason is as follows: Although there are actually two pairs of interdependent cyclic parameter links, they are identical except for the fact that they are located one level of recursion apart. If there exists a pair of infinite data transfer sequences without blockage at common parameter in any one pair of interdependent cyclic parameter links, this pair of infinite data transfer sequences can also exist in the other identical pair of cyclic parameter links. Since the cyclic connected data-link list in each cyclic parameter link represents the infinite data transfer sequence in this cyclic parameter link, we need to examine only one pair of cyclic connected data-link lists from one pair of interdependent cyclic parameter links. Actually, there are **three different situations:**

(1) all the involved interdependent cyclic parameter links have the same length,

(2) the lengths of the involved interdependent cyclic parameter links are different in a ratio of an exact multiple, and

(3) the lengths of the involved interdependent cyclic parameter links are in a ratio that cannot be reduced to an exact multiple.

Their graphical representations of these three situations appear in Figure 5.6. The example in Figure 5.5 is in the situation (1) and it is the simplest case. Since two interdependent cyclic parameter links have the same length, the common parameters in their corresponding cyclic connected data-link lists are located in the same level of recursion. This is clearly shown by the graphical representations in Figure 5.6. In the graphical representation of the interdependent cyclic parameter links of equal length, every common parameter in both cyclic parameter links can be paired up. This is indicated by the dotted lines between the pairs of squares. However, this is not true for interdependent cyclic parameter links with different lengths. For both the graphical representations of the interdependent cyclic parameter links with a ratio of an exact

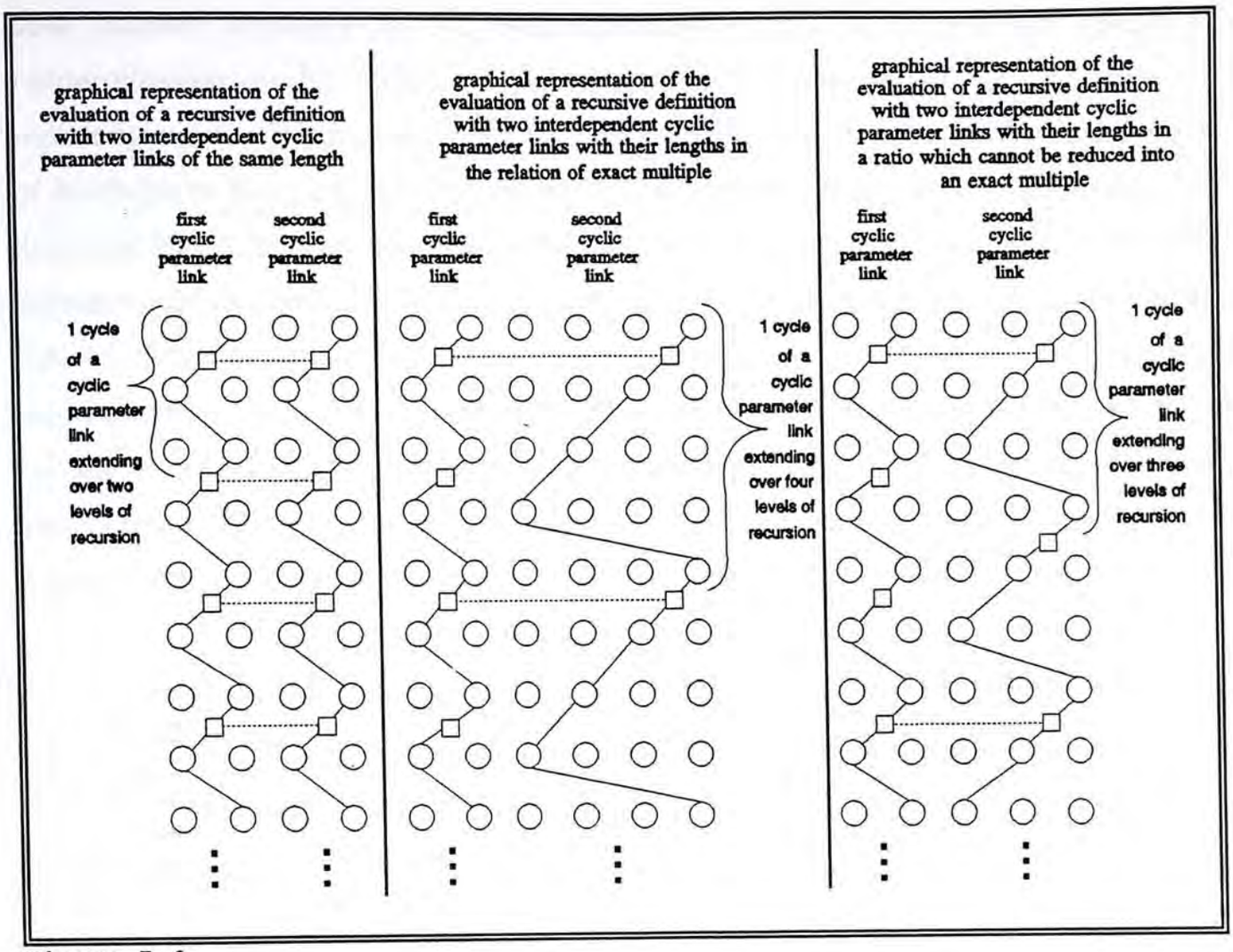


Figure 5.6

multiple or with a ratio that is not an exact multiple, there are some common parameters in one or both cyclic parameter links that cannot be connected to the common parameter in the other cyclic parameter link. Therefore, **in situation (2) and situation (3), there are more than one pair of interdependent connected data-link lists to be examined.** These situations will be discussed in Section 5.1.1.2.2 to see how the test of blockage at common parameter can be conducted in the more complicated situations.

Furthermore, the graphical representation of the recursive definition in Figure 5.5 shows us how nontermination occurs in a recursive definition with interdependent cyclic parameter links. If there is no blockage at common parameter in one of the repeating segments in an infinite data transfer sequence, there is no blockage at common parameter in other repeating segments as well. It means that there is no blockage at any point of the infinite data transfer sequence. If this is true for at least one infinite

data transfer sequence in the cyclic parameter links in a recursive definition, nontermination results. Since a cyclic connected data-link list actually represents the repeating segment of an infinite data transfer sequence, we only need to conduct the test of blockage to every common parameter in **one complete cycle of** the cyclic connected data-link list to find out whether any blockage can happen in **the entire data transfer sequence.** In the recursive definition in Figure 5.5, the graphical representation shows that each cyclic parameter link has a cyclic connected data-link list. The graphical representation also shows that there is no blockage at common parameter for a complete cycle of the cyclic connected data-link list in each cyclic parameter link. Therefore, an infinite data transfer sequence exists in both cyclic parameter links and consequently nontermination occurs.

Therefore, if we compare the example in Figure 5.5 with the case of interdependent cyclic and non-cyclic parameter links in Section 5.1.1.1, we can find that both illustrate the same phenomenon, that is, nontermination can occur if there is no blockage at all the common parameters on the cyclic connected data-link lists involved. However, the way to detect blockage at common parameter may be very different in these two cases. In the case of interdependent cyclic and non-cyclic parameter links, there is no constraint on the non-cyclic connected data-link lists involved since they represent the values passing through the non-cyclic parameter links. In the case of interdependent cyclic and non-cyclic parameter links, the **same** cyclic connected data-link list can share the common parameters with several **different** non-cyclic connected data-link list, yet the values transferring in a non-cyclic connected data-link list do not affect the values in another non-cyclic connected data-link list. On the contrary, in the case of interdependent cyclic parameter links, all connected data-link lists involved are **cyclic** connected data-link lists. Blockage at common parameter occurs at the cyclic parameter links involved **unless** the **same pair** of cyclic connected data-link lists can **share all the common parameters in every level of recursion.** In the following sections, we shall explore how interdependent cyclic parameter links and cyclic **connected data-link lists** with lengths in different ratios can complicate the situation and how one can test for blockage at common parameter in these situations.

5.1.1.2.1 Lengths of Cyclic Connected Data-link Lists in Different Ratios

Before we discuss the case of interdependent cyclic parameter links with lengths in different ratios, we shall first examine how the cyclic connected data-link lists of different lengths can affect the detection of blockage at common parameter. Similar to the case of interdependent cyclic parameter links, the lengths of the cyclic connected data-link lists can be classified into three categories:

(1) the lengths of the cyclic connected data-link lists in the different interdependent cyclic parameter links are the same,

(2) the lengths of the cyclic connected data-link lists in the different interdependent cyclic parameter links are different but in the ratio of an exact multiple, and

(3) the lengths of the cyclic connected data-link lists in the different interdependent cyclic parameter links are different and their lengths are in a ratio that cannot be reduced to an exact multiple.

To examine how the difference in the lengths of the cyclic connected data-link lists can complicate the detection of blockage at common parameter, we start with a simple case. In this section, we only consider the cyclic connected data-link lists with lengths in different ratios in those interdependent cyclic parameter links with equal length. Then in the next section we shall generalize the conclusion of our discussion to interdependent cyclic parameter links with different lengths.

In Figure 5.5, we already have an example of cyclic connected data-link lists with equal length. In the discussion of the example in Figure 5.5, we have also seen how nontermination can occur in such a situation. To detect nontermination in this case, we only need to slightly modify the data analysis method. Data analysis should be enhanced to also record the common parameter value sequence when constructing the cyclic connected data-link list in each of the interdependent cyclic parameter links. If the common parameter value sequences in different connected data-link lists are the same, blockage at common parameter does not occur and nontermination results. This modification on data analysis to handle the equal-length interdependent cyclic parameter links with equal-length cyclic connected data-link lists is simple and straightforward,

However, it becomes complicated if the cyclic connected data-link lists involved have different lengths.

In Figure 5.7, there is a recursive rule similar to the one in Figure 5.5 and there are four different sets of procedures. Similar to the recursive definition in Figure 5.5, there are two cyclic parameter links that both extend over two levels of recursion. The first one is located between parameters *AX/XA* and *BX/XB* while the second one is located between parameters *CX/XC* and *DX/XD*. With four different sets of procedures, four examples of interdependent cyclic parameter links are given. Each has interdependent cyclic parameter links of equal length sharing only one common parameter. However, in all four examples, the cyclic connected data-link list in the second cyclic parameter link extends over two cycles of the corresponding cyclic parameter link. Therefore, the lengths of the two cyclic connected data-link lists are different. When the lengths of the cyclic connected data-link lists are compared, we can see that their lengths are **in the ratio of an exact multiple** of 2. On the other hand, while the cyclic connected data-link list in the first cyclic parameter link only involves one common parameter, the one in the second cyclic parameter link involves two common parameters. By modifying the notion of cyclic connected data-link list used in Chapter 4, the common parameter values can be shown on the cyclic connected data-link list as well. The value indicated between *[]* is the value passing through the common parameter in this particular cyclic connected data-link list. It allows the difference between the cyclic connected data-link list from the first cyclic parameter link and the one from the second cyclic parameter link to be shown.

In Case (I), the cyclic connected data-link list formed in the first cyclic parameter link is *1--2--1*, with a length of 2, while the cyclic connected data-link list in the second cyclic parameter link is *11--12--13--14--11*, with a length of 4. So, the length of the cyclic connected data-link list in the second cyclic parameter link is double the length of the cyclic connected data-link list in the first cyclic parameter link. As shown by the graphical representation of Case (I), the cyclic connected data-link list *1--2--1* formed by the facts *link1a1(1,a)*, *link1a2(a,2)* and *link1b(2,1)* in the first cyclic parameter link can pass only the value *a* through the common parameter *CP* during the recursion.

goal(AX,BX,CX,DX) :- link1a1(AX,CP), link1a2(CP,XB), link1b(BX,XA),
link2a(CX,XD), link2b1(DX,CP), link2b2(CP,XC), goal(XA, XB, XC, XD).

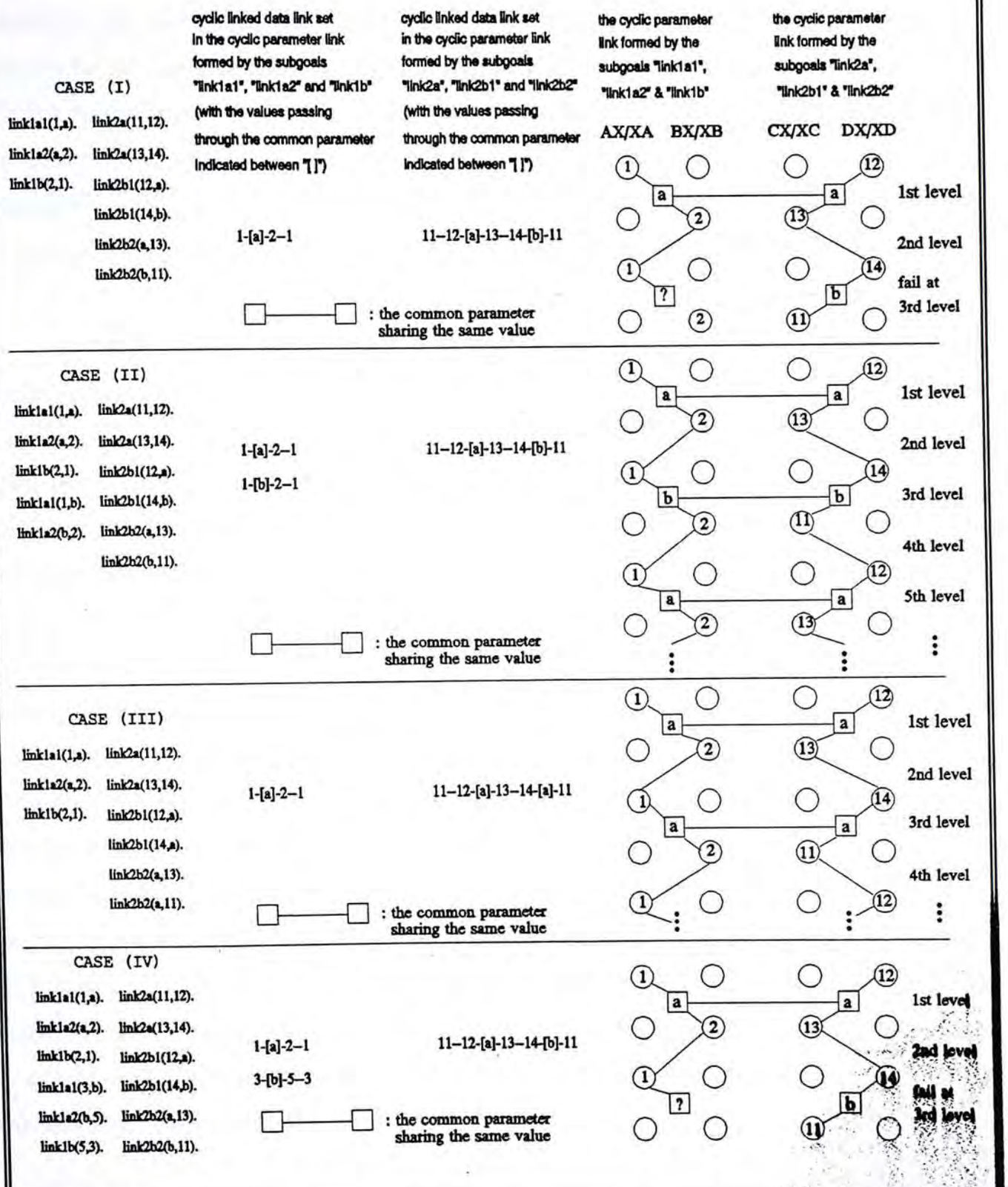


Figure 5.7

However, the cyclic connected data-link list in the second cyclic parameter link, as shown by the diagram in Figure 5.7, can complete a cycle if only both the values *a* and *b* are passed through the common parameter sequentially. At the third level of recursion, the cyclic connected data-link list in the first cyclic parameter link can allow only the value *a* to pass through the common parameter while the cyclic connected data-link list in the second cyclic parameter link can only allow the value *b* to pass through the common parameter. Such a conflict of value at common parameter causes the evaluation to stop at the subgoal *link2b1*. Therefore, nontermination is avoided in Case (I) due to the common parameter blockage at the third level of recursion.

In Case (II), the blockage at common parameter is eliminated because new facts are introduced to the procedures of the subgoals *link1a1* and *link1a2*. As shown in Figure 5.7, if we also consider the value passing through the common parameter, we have only one cyclic connected data-link list, *1-[a]-2--1* in Case (I), but two cyclic connected data-link lists, *1-[a]-2--1* and *1-[b]-2--1* in Case (II). The new cyclic connected data-link list *1-[b]-2--1* is formed because of the newly introduced facts. As indicated by the graphical representation of Case (II), the blockage is eliminated because the two cyclic connected data-link lists can be connected to form a longer cyclic connected data-link list of *1-[a]-2--1-[b]-2--1* with a length of four. This new cyclic connected data-link list is an **exact match** to the long cyclic connected data-link list *11--12-[a]-13--14-[b]-11* in the second cyclic parameter link **in terms of length and the sequence of values passing through the common parameter**. In Case (III), there is another way to eliminate the blockage. The cyclic connected data-link list in the second cyclic parameter link is modified to pass only value *a* through the common parameter. Although Case (II) and Case (III) demonstrate two different ways to allow **infinite data** transfer sequences to exist in the interdependent cyclic parameter links **(in which the** length of their cyclic connected data-link lists have a ratio of exact multiple**), they share the two same basic conditions:**

(1) the shorter cyclic connected data-link list can be connected to form a cyclic connected data-link list with a length the same as that of the longer cyclic connected data-link list, and

(2) the cyclic connected data-link list formed by the shorter cyclic connected data-link list and the longer cyclic connected data-link list have the same sequence of common parameter values.

In Case (I), although condition (1) is fulfilled, condition (2) cannot be met. The cyclic connected data-link list in the first cyclic parameter link has the common parameter value sequence of *a--a* while the one in the second cyclic parameter link has the sequence of *a--b*. Case (IV) again shows how the two conditions are needed for eliminating blockage at common parameter. In Case (IV), condition (1) cannot be satisfied and blockage occurs at the second common parameter shared between these two cyclic connected data-link lists. There are two cyclic connected data-link lists: *1-[a]-2--1* and *3-[b]-5--3*. Although each of them can match to one of the two values transferred through the common parameter by the long cyclic connected data-link list in the second cyclic parameter link, they cannot form a long cyclic connected data-link list that matches exactly the length of *11--12-[a]-13--14-[b]-11* as in Case (II). Because the value transferred through the common parameter is only *a* in the cyclic connected data-link list *1-[a]-2--1*, the conflict of values causes blockage to occur at the common parameter in the second half of the cyclic connected data-link list *11--12-[a]-13--14-[b]-11*, in which only the value *b* can be transferred through the common parameter. The graphical representation of Case (IV) shows how evaluation fails at the third level of recursion due to this conflict of values at the common parameter. Although the backtracking mechanism will cause other alternative paths to be tried after blockage occurs, the conflict of values will not be resolved in other alternative paths and will block further recursion.

The four cases in Figure 5.7 show the basic concept underlying the modification of data analysis to handle cyclic connected data-link lists with lengths in the ratio of an exact multiple. As shown in Chapter 4, the cause of nontermination is the presence of an infinite data transfer sequence in the cyclic parameter links in a recursive definition. On the other hand, if the blockage at common parameter is not considered, each cyclic connected data-link list found in a cyclic parameter link actually represents a repeating segment of an infinite data transfer sequence in this cyclic parameter link. Therefore,

data analysis can be adapted to the case of interdependent cyclic parameter links with the addition of a process to verify whether a cyclic connected data-link list detected by data analysis can or cannot form any blockage at common parameter. In the case of **equal length** cyclic connected data-link lists as in the example in Figure 5.5, **comparing** the common parameter value sequences in all the involved cyclic connected data-link lists **directly** can verify whether any blockage at common parameter occurs. However, in the case of cyclic connected data-link lists with **lengths in the ratio of an exact multiple** (as shown by the examples in Figure 5.7) the common parameter value sequences from the cyclic connected data-link lists with different lengths **cannot be compared directly** because the lengths of these common parameter value sequences are also different. However, as shown by Case (II) and Case (III) in Figure 5.7, a cyclic connected data-link list can be linked to itself or another cyclic connected data-link list with certain appropriate values to form a longer cyclic connected data-link list. As suggested by the four examples in Figure 5.7, we can link the shorter cyclic connected data-link lists to form a cyclic connected data-link list with a length equal to that of the longer cyclic connected data-link list. Then the common parameter value sequences in both the cyclic connected data-link list (constructed from the shorter ones) and the longer cyclic connected data-link list have the same length and they can be compared directly. If nontermination occurs, there will exist a repeating segment in an infinite data transfer sequence which has the **same common parameter value sequence** in each of the involved interdependent cyclic parameter links. Since **the cyclic connected data-link list constructed out of the shorter ones can also represent a repeating segment of an infinite data transfer sequence**, the absence of any blockage at common parameter is indicated if the common parameter value sequence from the longer cyclic connected data-link list is equal to the sequence from the cyclic connected data-link list constructed out of the shorter ones.

In conclusion, the adaptation of data analysis for the **case of equal-length** interdependent cyclic parameter links with the lengths of their cyclic **connected data-link** lists in the ratio of an exact multiple is as follows: According **to the ratio, the shorter** cyclic connected data-link list is connected to itself or another **one with the same length** in the same cyclic parameter link for a number of times as **indicated by the ratio. For**

example, if the cyclic connected data-link lists from the interdependent cyclic parameter links CP_1 and CP_2 are in the ratio of **3:1**, the cyclic connected data-link list in the cyclic parameter link CP_2 must be connected to itself three times while the cyclic connected data-link list in CP_1 is unchanged. Then the common parameter value sequences of these equal-length cyclic connected data-link lists are compared. If the same common parameter value sequence can be found in cyclic connected data-link lists from different interdependent cyclic parameter links, there is no blockage at common parameter between a certain pair of infinite data transfer sequences and nontermination will happen. If cyclic connected data-link lists from more than two interdependent cyclic parameter links are involved, their lengths are considered to be in the ratio of an exact multiple only if **every pair** of the cyclic connected data-link lists have lengths in the ratio of an exact multiple. For example, suppose that there are five cyclic parameter links, say, CP_1, CP_2, CP_3, CP_4 and CP_5 with the lengths of their cyclic connected data-link lists in the ratio of **1:2:4:8:16** respectively. Their cyclic connected data-link lists are in the ratio of an exact multiple since the lengths of any pair is in the ratio of an exact multiple. However, if the lengths of the cyclic connected data-link lists of CP_1, CP_2, CP_3, CP_4 and CP_5 are in the ratio of **1:2:4:6:8** respectively, the ratio between the lengths of the cyclic connected data-link lists from the cyclic parameter links CP_4 and CP_5 is **2:3**, which is not in the ratio of an exact multiple. Because there is one pair of cyclic parameter links with lengths that cannot be reduced to an exact multiple, there is a shorter cyclic connected data-link list which cannot construct a new cyclic connected data-link list with the length equal to the length of the longest one. In this example, the cyclic connected data-link list in the cyclic parameter link CP_4 (with the length of 6) cannot form a cyclic connected data-link list with the length of 8.

Similarly, data analysis can be adapted to the case of cyclic connected data-link lists with their lengths **not in the ratio of an exact multiple** by constructing cyclic connected data-link lists of equal length in every involved cyclic parameter link from the shorter cyclic connected data-link lists in these cyclic parameter links. The method is just the same as that discussed above: for every interdependent cyclic parameter link, find an appropriate repeating segment of an infinite data transfer sequence in which a common parameter value sequence has its length equal to the sequences from other

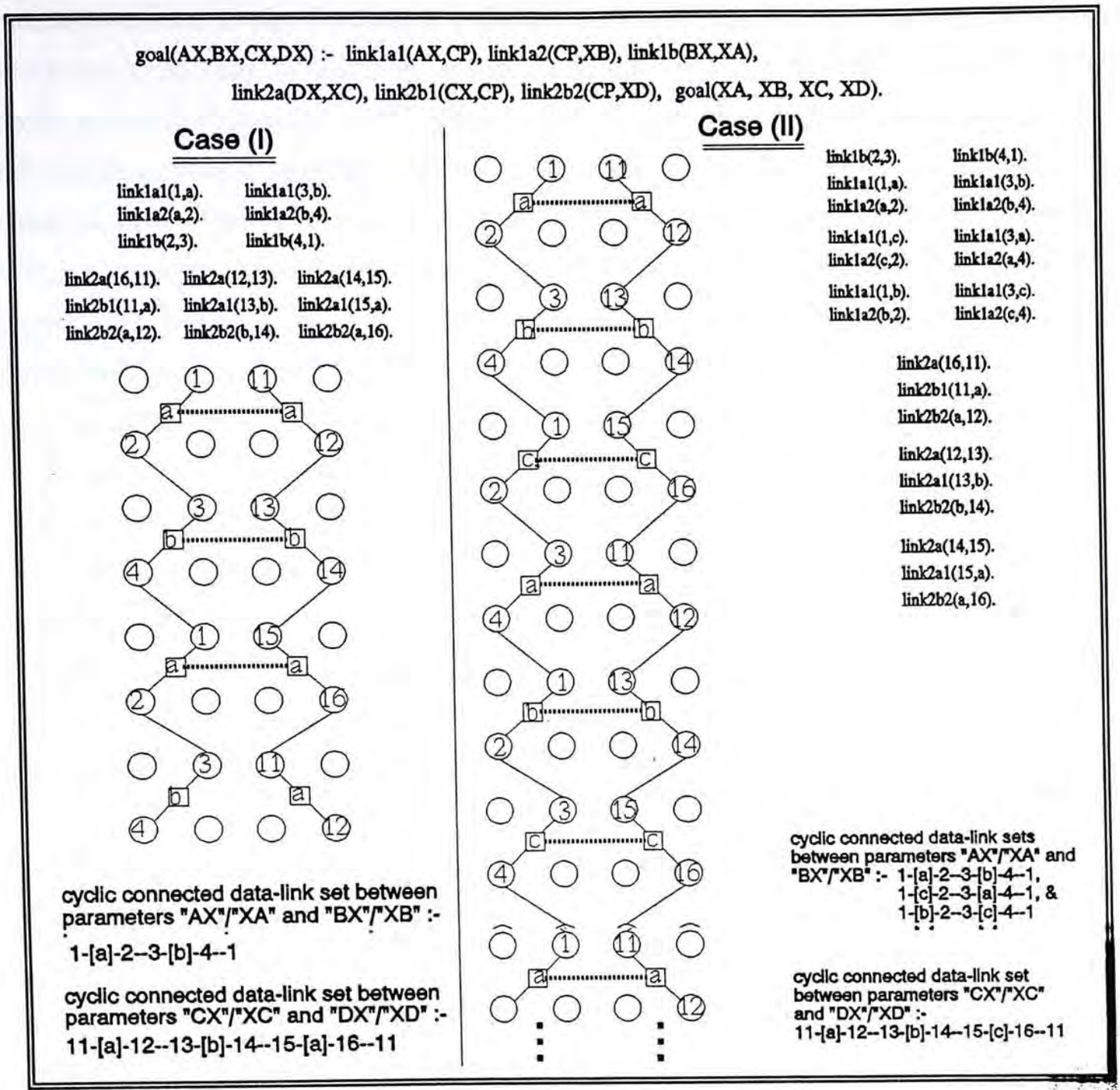


Figure 5.8

cyclic parameter links so that it can be compared directly to them. By contrasting Case (I) and Case (II) in Figure 5.8, we can see how one can find the appropriate repeating segment: it is tantamount to constructing an equal-length cyclic connected data-link list in all the involved cyclic parameter links from the shorter cyclic connected data-link lists. As shown by the graphical representation of Case (I), **the absence of blockage at common parameter is not guaranteed even though we cannot find blockage in one complete cycle of both** long and short cyclic connected data-link lists. In the graphical representation of Case (I), we can see that no blockage at common parameter occurs at the first three levels of recursion. Since the shorter cyclic connected data-link list only

extends over two levels of recursion while the longer one extends over only three levels, there is no blockage at common parameter at the first cycle of each long and short cyclic connected data-link lists. Because **the lengths of a repeating segment** of an infinite data transfer sequence represented by a cyclic connected data-link list must be **equal to, or a multiple of the length of this cyclic connected data-link list,** for the first cyclic parameter link between the parameters *AX/XA* and *BX/XB*, the repeating segment of the infinite data transfer sequence represented by this shorter cyclic connected data-link list can only have the length of **4** or **8** but not **6**. Therefore, the absence of blockage at common parameter in a segment with the length of **6** in the first cyclic parameter link does not guarantee the absence of blockage in a repeating segment of the entire infinite data transfer sequence in the first cyclic parameter link. However, blockage at common parameter is surely absent only if there is a common parameter value sequence shared by the repeating segments of the infinite data transfer sequences in every involved cyclic parameter link. In Case (I), therefore, the same common parameter value sequence is not truly shared by the repeating segments of two infinite data transfer sequences from the two involved cyclic parameter links.

On the other hand, Case (II) shows how a common parameter value sequence can be truly shared by the repeating segments of two infinite data transfer sequences from the interdependent cyclic parameter links. By the graphical representation of Case (II), we can see that nontermination occurs if the same common parameter value sequence is shared by two cyclic connected data-link list with equal length and both are constructed out of some shorter cyclic connected data-link lists. The cyclic connected data-link list formed by linking up some shorter cyclic connected data-link lists can also represent a repeating segment of an infinite data transfer sequence. Therefore, the cyclic connected data-link lists constructed out of the shorter ones in both cyclic parameter links in the graphical representation of Case (II) represent the repeating segment with equal length in both cyclic parameter links. And these two repeating segment have the common parameter value sequences of equal length as well. Then the two sequences can be compared directly to determine whether nontermination will occur.

In other words, in the case of cyclic connected data-link lists with lengths not in a ratio of an exact multiple, the common parameter value sequence to be compared needs to come from the equal length cyclic connected data-link lists from different interdependent cyclic parameter links, and these equal cyclic connected data-link lists are formed by connecting the shorter cyclic connected data-link lists. Therefore, the resulting cyclic connected data-link lists from the shorter ones must have a length equal to the **least common multiple** of the lengths of the involved cyclic connected data-link lists. For example, in Case (II), since the length of the cyclic connected data-link list in the first cyclic parameter link between *AX/XA* and *BX/XB* is **4**, while the length of the cyclic connected data-link list in the second cyclic parameter link between *CX/XC* and *DX/XD* is **6**, the length of the new cyclic connected data-link list must be the least common multiple of **4** and **6**, which is **12**. This is confirmed by the graphical representation of Case (II) in Figure 5.8. Therefore, we only need to compare the common parameter value sequences which come from the cyclic connected data-link lists formed by repeating the cyclic connected data-link list in the first cyclic parameter link three times and the one in the second cyclic parameter link twice.

In conclusion, for cyclic connected data-link lists with lengths in different ratios, the adaptation of data analysis to the case of interdependent cyclic parameter links is to include the comparison among the common parameter value sequences from the repeating segment of the infinite data transfer sequence from different interdependent cyclic parameter links. If there is a common parameter value sequence shared by the repeating segments of all involved cyclic parameter links, we can be sure about the absence of any blockage at common parameter and the occurrence of nontermination during the evaluation of the corresponding recursive definition. Data analysis can give us the repeating segment of the infinite data transfer sequence in each cyclic parameter link by detecting the cyclic connected data-link list in each cyclic parameter link. **If data analysis is slightly modified to also provide the values passing through the common parameter** when constructing the cyclic connected data-link list, we can also have the common parameter value sequence of the repeating segment of the infinite data transfer sequence represented by this connected data-link list.

If the involved connected data-link lists have the same length, the common parameter value sequence from the connected data-link lists can be directly compared. On the other hand, if the cyclic connected data-link lists are of different lengths, we need to construct some new connected data-link lists from the involved cyclic parameter links before the comparison. By connecting the shorter cyclic connected data-link lists to itself or to some cyclic connected data-link lists with appropriate values to form a longer cyclic connected data-link list with a length equal to the least common multiple of the lengths of all the involved cyclic connected data-link lists, the appropriate connected data-link lists can be formed. Then the common parameter value sequences from these new connected data-link lists can be compared to detect blockage at common parameter. If we compare the discussion on the cases in Figure 5.7 with the discussion on the cases in Figure 5.8, we can see that **the case of equal length and the case of lengths in a ratio of an exact multiple are in fact special cases of the more general case of cyclic connected data-link lists with their lengths not in a ratio of an exact multiple.** On the one hand, in the case where the lengths are in a ratio of an exact multiple, the least common multiple is always equal to the length of the longest cyclic connected data-link list. On the other hand, in the case of equal length, the least common multiple is always equal to the original length of all the involved cyclic connected data-link lists. Therefore, we can summarize the method adaptation required to handle the interdependent cyclic parameter links with all equal-lengths cyclic parameter links as follows:

(1) If any cyclic connected data-link list is detected in every interdependent cyclic parameter link, examine the cyclic connected data-link lists by following the steps (2) to (7); otherwise, no test of blockage at common parameter is needed.

(2) Find the common parameter value sequences of each cyclic connected data-link list in each interdependent cyclic parameter link.

(3) Find the least common multiple of the length of all involved cyclic connected data-link lists.

(4) Connect the original cyclic connected data-link list (formed by data analysis) in each cyclic parameter link to some longer cyclic connected data-link lists with length equal to the least common multiple.

(5) Find out the common parameter value sequence of each cyclic connected data-link list constructed in step (4).

(6) Compare the common parameter value sequences of the new cyclic connected data-link lists constructed by the step (4).

(7) If at least one cyclic connected data-link list formed by steps (3) and (4) in all the interdependent cyclic parameter links share the same common parameter value sequence, there is no blockage at common parameter and nontermination is detected; otherwise, nontermination will not occur.

5.1.1.2.2 Cyclic Parameter Links with Lengths in Different Ratios

Because of the presence of multi-level cyclic parameter links, as shown by the graphical representations in Figure 5.6, there are three ways to classify interdependent cyclic parameter links in terms of their lengths. First, the interdependent cyclic parameter links can all have the same length, i.e., all extend to the same number of levels of recursion. The interdependent cyclic parameter links in Figure 5.2 and Figure 5.5 are examples. Second, interdependent cyclic parameter links have different lengths but their lengths are in the ratio of an exact multiple. The interdependent cyclic parameter links in the recursive rule below is an example of this case:

```
rule1(AX, BX, CX, DX, EX, FX):-
        link1a1(BX, CP), link1a2(CP, XA), link1b(AX, XB),
          link2a(CX, XF), link2b(DX, XC), link2c(EX, XD),
          link2d1(FX, CP), link2d2(CP, XE),
                    rule1(XA, XB, XC, XD, XE, XF).
```

We can find that the subgoals *link1a1*, *link1a2* and *link1b* actually form a cyclic parameter link which extends over two levels of recursion while the cyclic parameter link formed by the subgoals *link2a*, *link2b*, *link2c*, *link2d1* and *link2d2* has a length of four levels of recursion. Therefore the length of the cyclic parameter link formed by the subgoals *link2a*, *link2b*, *link2c*, *link2d1* and *link2d2* is exactly twice of the length of the one formed by the subgoals *link1a1*, *link1a2* and *link1b*. The graphical representation

at the center in Figure 5.6 can be the graphical representation of the above recursive rule. In the case where more than two cyclic parameter links are involved, (which is similar to the case of more than two cyclic connected data-link lists with their lengths in the ratio of an exact multiple), we can consider their lengths to be in the ratio of an exact multiple if the ratio of the length of **every pair** of the cyclic parameter links is an exact multiple. The reason is the same as that for the lengths of cyclic connected data-link lists in the ratio of an exact multiple discussed in the previous section. Third, the interdependent cyclic parameter links involved have different lengths and the ratio of their lengths is not an exact multiple. The interdependent cyclic parameter links in the recursive rule below is an example:

```
rule2(AX, BX, CX, DX, EX):-
        link1a1(BX, X), link1a2(X, XA), link1b(AX, XB),
                link2a(DX, XC), link2b(CX, XE),
                link2c1(EX, X), link2c2(X, XD),
                        rule2(XA, XB, XC, XD, XE).
```

Again, the length of the first cyclic parameter link formed by *link1a1*, *link1a2* and *link1b* is two levels of recursion. But the length of the second cyclic parameter link formed by *link2a*, *link2b*, *link2c1* and *link2c2* is three levels. Therefore their lengths are in the ratio of 2:3 which is not an exact multiple. This can be represented by the graphical representations at the right hand side in Figure 5.5. We can see that the cycle of the first cyclic parameter link overlap with the cycle of the second cyclic parameter link.

Because a common parameter is shared by all the interdependent cyclic parameter links, **we cannot consider each cyclic parameter link separately** as in the case of independent cyclic parameter links. However, as shown by the graphical representations in Figure 5.6, the interdependent cyclic parameter links with lengths in different ratios can share the common parameter in different ways. For two interdependent cyclic parameter links of the same length, they share every common parameter. In the graphical representation of the equal-length interdependent cyclic parameter links in Figure 5.6, we can clearly see that all the common parameters in one

cyclic parameter link can be paired up with a line to the common parameter in another cyclic parameter link at the same level of recursion. However, we can find that some common parameters cannot be shared between two interdependent cyclic parameter links if their lengths are not equal. The graphical representations in the center and at the right hand side show that some squares cannot be connected to the other squares by a line. In fact, there is simply no other square in the other cyclic parameter link that is at the same level of recursion. In the case where their lengths are in the ratio of an exact multiple, every common parameter in the longer cyclic parameter link can always be paired up with a common parameter in the shorter cyclic parameter link but not vice versa. In the case of their lengths in a ratio which is not an exact multiple, both cyclic parameter links have some common parameters cannot be paired up.

the expanded version of the graphical representation in Figure 5.6 of a recursive definition with two interdependent cyclic parameter links with their lengths in a ratio which cannot be reduced into an exact multiple

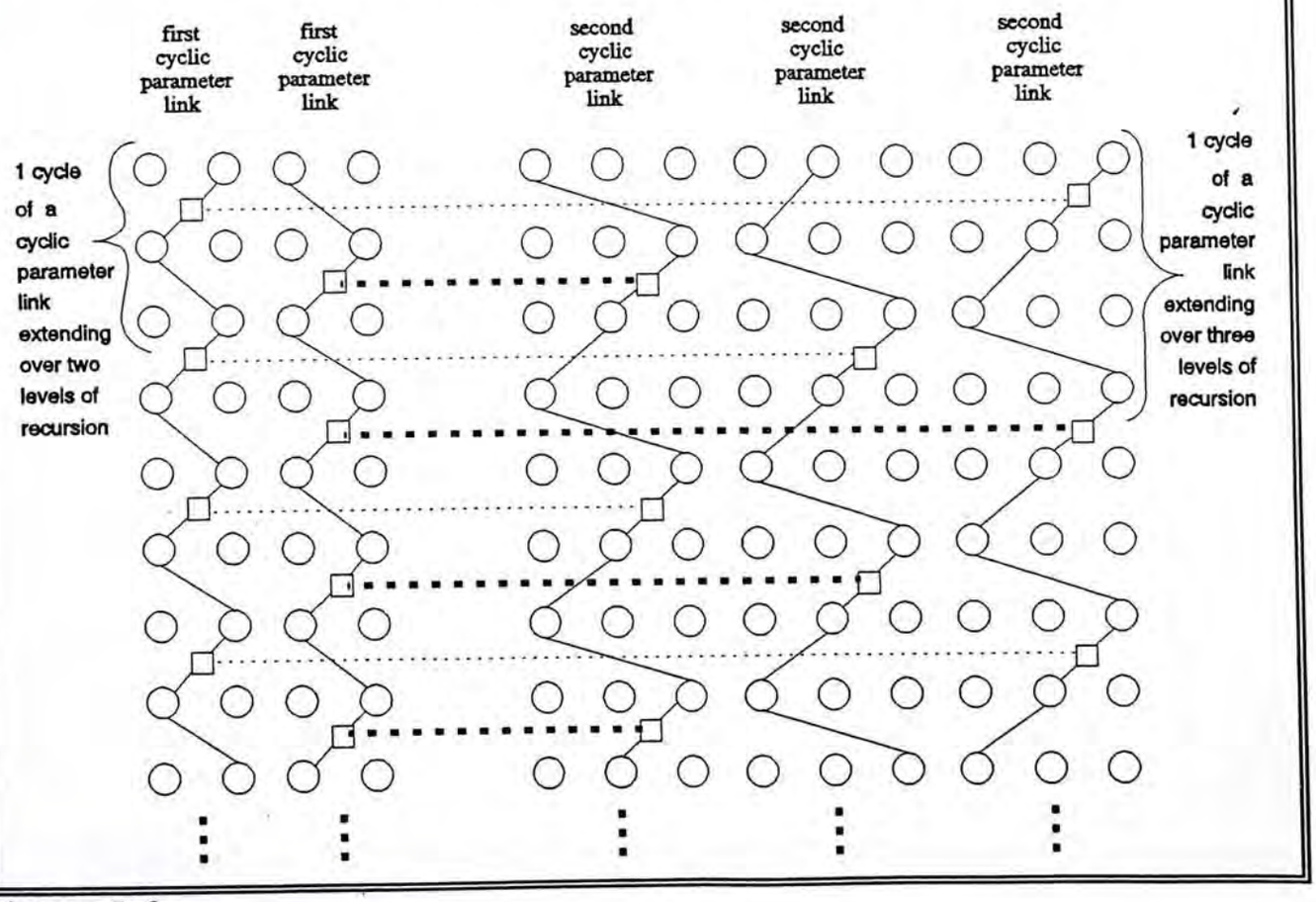


Figure 5.9

However, **the lack of paired up common parameters at certain points shown by the graphical representations in Figure 5.6 does not mean that the corresponding interdependent cyclic parameter links must have data transfer blockage at these points.** The graphical representations in Figure 5.6 only show one pair of all the possible cyclic parameter links. If Figure 5.9 is compared with the graphical representation at the right hand side in Figure 5.6, we can see that both graphical representations on the left hand side of Figure 5.6 and Figure 5.9 are the graphical representation of the above recursive rule *rule2*, except that the one in Figure 5.9 is an expanded version of the one in Figure 5.6. In Figure 5.9, all the cyclic parameter links are shown. As explained in Section 5.1.1.2, in order to form a cyclic parameter link extending over two levels between the parameter *AX/XA* and *BX/XB*, two identical parameter links must exist in the same set of parameters. Similarly, three identical cyclic parameter links are also present in the parameters *CX/XC*, *DX/XD* and *EX/XE*. Therefore, there are **two groups** of interdependent cyclic parameter links instead of two interdependent cyclic parameter links. If the above recursive rule is considered, it is easier to see that some values must be transferred through the common parameter *X* in every level of recursion when nontermination occurs. If the graphical representation in Figure 5.6 is considered, it is hard to see how some values can be transferred through the common parameter during the recursion. The graphical representation in Figure 5.6 seems to suggest that some common parameter cannot be shared between the interdependent cyclic parameter links since it shows that some squares in both cyclic parameter links cannot be paired up with other squares. This implies blockage at common parameter and no nontermination will happen. But Figure 5.9 reveals that all common parameters can actually be paired up with other common parameters if all the possible cyclic parameter links are considered. The interaction does not exist between two cyclic parameter links but in fact between two groups of cyclic parameter links. The graphical representation in Figure 5.9 indicates how the connected data-link lists with different lengths can interact with each other at the common parameters in different levels of recursion.

Therefore, there is a basic difference between the ways to handle interdependent cyclic parameter links with equal length and those with different lengths. **In the case of equal length,** for every involved cyclic parameter link, the locations of the common

parameters in a cyclic parameter link are also the same. Therefore, every cyclic connected data-link list in these interdependent cyclic parameter links have their common parameter in the same level of recursion. In this situation, **for every group of cyclic parameter links formed among the same set of parameters, only one of them (rather than the whole group) requires consideration.** On the other hand, **in the case of the interdependent cyclic parameter links with different lengths,** as shown by Figure 5.9, **more than one cyclic parameter link in each group of cyclic parameter links among the same set of parameters should be our concern.** As what has been shown by the graphical representation in Figure 5.9, all the two identical cyclic parameter links formed between the parameters *AX/XA* and *BX/XB* and all the three identical cyclic parameter links among the parameters *CX/XC*, *DX/XD* and *EX/XE* all act together to allow no blockage to exist at all common parameters appearing in the infinite data transfer sequences. In the following sections, we shall see how one can handle the two different cases of interdependent cyclic parameter links with different lengths: lengths in the ratio of an exact multiple and lengths not in the ratio of an exact multiple. However, since they are rare cases in Prolog programming, we only discuss the basic concept of how to detect blockage at common parameter without too many unnecessary details.

In Figure 5.9, the graphical representation describes interdependent cyclic parameter links with lengths in a ratio that cannot be reduced to an exact multiple. It shows how the two groups of cyclic parameter links can share the common parameters during the different levels of recursion. **In each particular level of recursion, a common parameter must appear in one of all the cyclic parameter links in each group of involved cyclic parameter links.** By comparing the values transferring through the common parameter in each group of cyclic parameter links, we can determine whether there is any blockage occurs. As suggested in the previous discussion, we can know whether blockage at common parameter appears in the entire infinite data transfer sequences of the involved cyclic parameter links by examining a repeating segment of each of these infinite data transfer sequences. Therefore, if there is a repeating segment of a data transfer sequence appearing in each group of cyclic parameter links and **the common parameters from different groups of cyclic parameter links share the same**

value at every level of recursion in the repeating segment, there is no blockage at common parameter and nontermination will result.

As discussed, though the lengths of the different groups of cyclic parameter links are different, we can find repeating segments of the infinite data transfer sequences in every cyclic parameter link with the same length. Because their lengths are equal, they represent one complete cycle of infinite data transfer sequences in all the involved cyclic parameter links. If there is no blockage in this cycle of all the involved infinite data transfer sequences, we can be sure that there is no blockage at all. However, in order to determine the length of the repeating segment in the case of interdependent cyclic parameter links with their lengths in the ratio of an exact multiple, we must also consider the relation between the length of the cyclic connected data-link lists and the length of their corresponding cyclic parameter links. Because a cyclic connected data-link list in fact represents a repeating segment of an infinite data transfer sequence, the length of cyclic connected data-link list is crucial in determining the length of the repeating segment of any infinite data transfer sequence in a cyclic parameter link. However, it is obvious that **the length of a cyclic connected data-link list can either be equal to or a multiple of the length of its corresponding cyclic parameter link** (because an infinite data transfer sequence is formed by the values that can pass through one or several complete cycles of a cyclic parameter link). If they are the same, the lengths of the cyclic connected data-link lists of the involved interdependent cyclic parameter links must be in a ratio that cannot be reduced to an exact multiple. On the other hand, if they are not the same, the lengths of the involved cyclic connected data-link lists can be in one of the three relations: equal length, different length in the ratio of an exact multiple or not in an exact multiple, just as what has been described in Section 5.1.1.2.1. However, as also shown in Section 5.1.1.2.1, the cyclic connected data-link lists with different lengths can become the cyclic connected data-link lists of equal length by reflexive connecting several times. The length of the longer cyclic connected data-link lists constructed out of the shorter ones is equal to the least common multiple of the lengths of all these shorter cyclic connected data-link lists. Therefore, by connecting the shorter cyclic connected data-link list to form the cyclic connected data-link list with its length equal to the least common multiple of the lengths of all involved cyclic connected

data-link lists, we can find that the cyclic connected data-link list represents the repeating segment that has equal length in every group of involved cyclic parameter links. So the modification on data analysis to determine the presence of blockage at common parameter can be summarized as follows:

(1) Find the least common multiple of the lengths of the cyclic connected data-link lists present in every interdependent cyclic parameter link.

(2) For every interdependent cyclic parameter links, construct some cyclic connected data-link lists with a length equal to the least common multiple obtained by step (1).

(3) For each cyclic connected data-link list constructed by step (2), find the value used in the common parameter in each level of recursion. If no common parameter is used in a particular level, mark the corresponding level. Put the marks and values together in the order of their appearance in different levels of recursion to form a common parameter value sequence.

(4) Combine the common parameter value sequence of the same cyclic connected data-link list or other cyclic connected data-link lists from the same cyclic parameter link to form a common parameter value sequence without any mark. If two common parameter value sequence have values at the same level of recursion, shift one of them a number level up or down and try again.

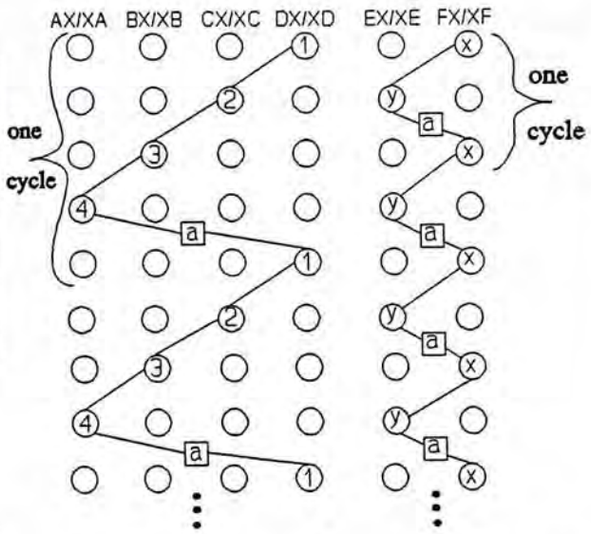
(5) Compare the common parameter value sequence constructed in step (4). If at least one common parameter value sequence is shared by the cyclic connected data-link lists from every cyclic parameter link, the absence of blockage at common parameter is confirmed. Nontermination will result.

This modification may be very time-consuming in some situations. However, the above steps only serve as a summary of our above discussion. Since rarely do we have a recursive definition with interdependent cyclic parameter links in different lengths and it is even more rare for their lengths to be non-reducible to an exact multiple, we do not go into further detail in this work. Actually, it is not only hard to determine the presence of blockage at common parameter in this case, it is also hard to understand a Prolog program with any interdependent cyclic parameter links with lengths not in the ratio of an exact multiple.

Figure 5.10 shows an example of interdependent cyclic parameter links with lengths in the ratio of an exact multiple and their graphical representations. The graphical representation at the bottom of Figure 5.10 shows how the common parameter

```
goal(AX,BX,CX,DX,EX,FX) :-
      link1a(BX,XA), link1b(CX,XB), link1c(DX,XC),
      link1d1(AX,CP), link1d2(CP,XD),
        link2a(FX,XE), link2b1(EX,CP), link2b2(CP,XF),
            goal(XA,XB,XC,XD,XE,XF).

link1a(3,4).     link1b(2,3).     link1c(1,2).
link1d1(4,a).    link1d2(a,1).

link2a(x,y).     link2b1(y,a).    link2b2(a,x).
```

the graphical representation below showing how the same pair of data transferring sequences in two interdependent cyclic parameter links with their lengths in a relation of exact multiple can share the same value at the common parameter during every levels of recursion



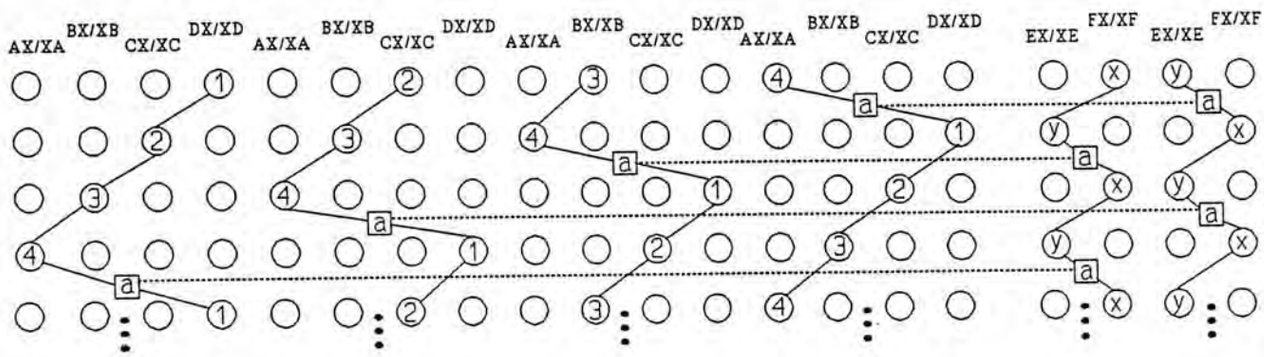


Figure 5.10

in every level of recursion is shared between the two groups of cyclic parameter links. At first glance, it is similar to the case of interdependent cyclic parameter links with their lengths not in the ratio of an exact multiple. If only two cyclic parameter links instead of two groups of cyclic parameter links are considered, there are some levels of recursion with a common parameter appearing in only one cyclic parameter link. However, there is a significant difference between the two cases of different length interdependent cyclic parameter links. In the case of the cyclic parameter links with lengths not in the ratio of an exact multiple, when only two cyclic parameter links instead of two groups of cyclic parameter links are considered, **both cyclic parameter links have common parameters in only one of the two cyclic parameter links that cannot be paired up by the common parameter in the other cyclic parameter link. The whole** group of cyclic parameter links among the same set of parameters is **required to allow** all common parameters to be paired up. But in the case of **interdependent cyclic**

parameter links in the ratio of an exact multiple, we can see that **all common parameters in the longer cyclic parameter link can be paired up by the common parameters from the shorter cyclic parameter link.** In the graphical representation at the bottom of Figure 5.10, we can find that only two out of the group of four longer cyclic parameter links can provide the common parameters to pair up all common parameters in the shorter cyclic parameter link. **Moreover, the cyclic connected data-link lists in these two longer cyclic parameter links are identical.** Therefore, it shows that, in the case of an exact multiple ratio, there are certain situation in which we only need to consider two cyclic parameter links instead of two groups of cyclic parameter links when detecting the blockage at common parameter. The modifications to the method of data analysis can be summarized as follows:

(1) If the lengths of the interdependent cyclic parameter links are in the ratio of an exact multiple and the length of their cyclic connected data-link lists are equal to or in a multiple of the lengths of the cyclic parameter links, find the common parameter value sequences of each involved cyclic connected data-link list as in step (2) below.

(2) If, at certain level of recursion, no common parameter is involved to transfer value, put a mark on the common parameter value sequence; otherwise, put the value used in the common parameter into the sequence.

(3) Compare the common parameter value sequences of the cyclic connected data-link list from different interdependent cyclic parameter links. If there at least one common parameter value sequence is shared among all the cyclic parameter links, nontermination will occur.

In this example, the lengths of the cyclic connected data-link lists and the lengths of cyclic parameter links are equal in all the involved interdependent cyclic parameter links so that the ratio of the lengths of their cyclic connected data-link lists are also in an exact multiple. Moreover, the length of the cyclic connected data-link list can relate to the length of the cyclic parameter link in different ways to allow the lengths of different cyclic connected data-link lists to form different ratios. In Figure 5.11, there is an example of interdependent cyclic parameter link which have lengths in the ratio of an exact multiple but the lengths of their cyclic connected data-link lists are equal. In Figure 5.12, the lengths of their cyclic connected data-link list are not in a ratio of an

```
goal(AX,BX,CX,DX,EX,FX) :-
   link1a(BX,XA), link1b(CX,XB), link1c(DX,CX),
      link1d1(AX,CP), link1d2(CP,XD),
         link2a(FX,XE), link2b1(EX,CP), link2b2(CP,XF),
            goal(XA,XB,XC,XD,XE,XF).
```

Case (I)

```
link1a(13,14).   link1b(12,13).   link1c(11,12).
link1d1(14,b).   link1d2(b,11).
link1a(23,24).   link1b(22,23).   link1c(21,22).
link1d1(24,a).   link1d2(a,21).

link2a(x,y).     link2b1(y,a).    link2b2(a,z).
link2a(z,w).     link2b1(w,b).    link2b2(b,x).
```

Case (II)

```
link1a(23,24).   link1b(22,23).   link1c(21,22).
link1d1(24,a).   link1d2(a,21).

link2a(x,y).     link2b1(y,a).    link2b2(a,z).
link2a(z,w).     link2b1(w,a).    link2b2(a,x).
```

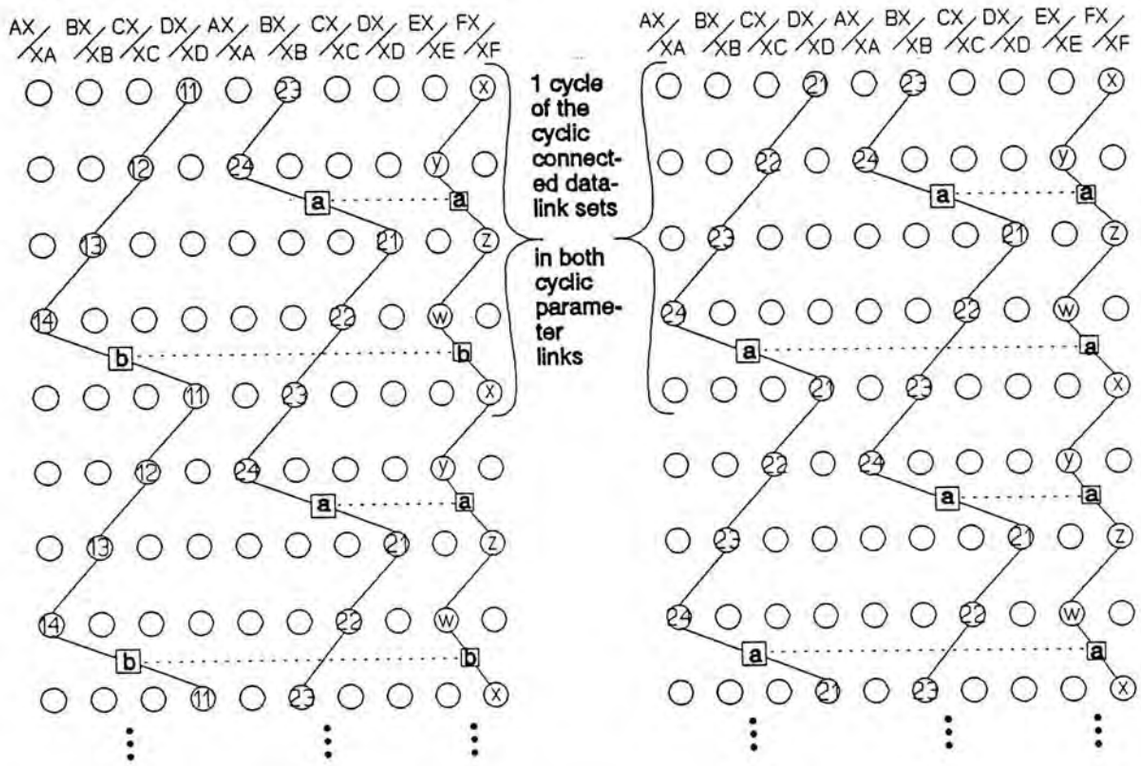


Figure 5.11

exact multiple.

In Figure 5.11, the graphical representations of both Case (I) and Case (II) show how the cyclic connected data-link list in the shorter cyclic parameter link can have the same length as the length of the cyclic connected data-link list in the longer cyclic parameter link. In the shorter cyclic parameter link, the length of the cyclic connected data-link list is twice the length of the cyclic parameter link. Therefore, in the shorter cyclic parameter link, **a cyclic connected data-link list in fact represents a repeating segment of an infinite data transfer sequence that passes through two cycles of the**

```
goal(AX,BX,CX,DX,EX,FX) :-
    link1a(BX,XA), link1b(CX,XB), link1c(DX,CX),
      link1d1(AX,CP), link1d2(CP,XD),
        link2a(FX,XE), link2b1(EX,CP), link2b2(CP,XF),
          goal(XA,XB,XC,XD,XE,XF).
```

Case (I)

```
link1a(13,14).   link1b(12,13).   link1c(11,12).
link1d1(14,b).   link1d2(b,11).
link1a(23,24).   link1b(22,23).   link1c(21,22).
link1d1(24,a).   link1d2(a,21).
link2a(x,y).     link2b1(y,a).    link2b2(a,z).
link2a(z,w).     link2b1(w,b).    link2b2(b,p).
link2a(p,q).     link2b1(q,c).    link2b2(c,x).
```

Case (II)

```
link1a(23,24).   link1b(22,23).   link1c(21,22).
link1d1(24,a).   link1d2(a,21).
link1d1(24,b).   link1d2(b,21).
link1d1(24,c).   link1d2(c,21).
link2a(x,y).     link2b1(y,a).    link2b2(a,z).
link2a(z,w).     link2b1(w,b).    link2b2(b,p).
link2a(p,q).     link2b1(q,c).    link2b2(c,x).
```

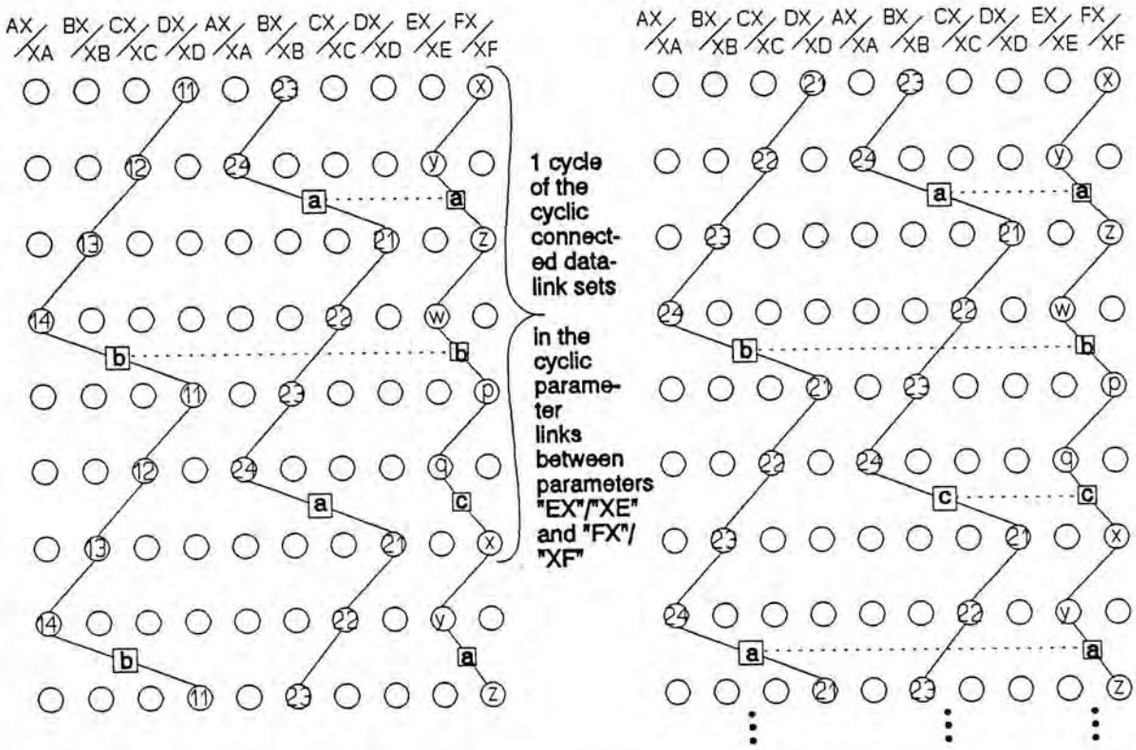


Figure 5.12

cyclic parameter link. Since a common parameter is involved in each cycle of the cyclic parameter link, two common parameters are involved in the cyclic connected data-link list in the shorter cyclic parameter link. In Case (I) in Figure 5.11, the graphical representation of the shorter cyclic parameter link between the parameters *EX/XE* and *FX/XF* shows this clearly. On the other hand, the longer cyclic parameter link among the parameters *AX/XA*, *BX/XB*, *CX/XC* and *DX/XD* has a cyclic connected data-link list only involved one common parameter. Because the cyclic connected data-link list in the longer cyclic parameter link can only involve one common parameter, it can only

share one of the two common parameters involved in the shorter cyclic parameter link. If the values passing through the two involved common parameters in the shorter cyclic parameter link are different, as in the example of Case (I) in Figure 5.11, it takes two cyclic connected data-link lists with different values passing through the common parameter to avoid any blockage at common parameter. The values passing through the common parameter in these two cyclic connected data-link lists must be the same as the values passing through the two common parameters in the cyclic parameter link between *EX/XE* and *FX/XF*. However, if the values passing through the two common parameters involved in the cyclic connected data-link list in the shorter cyclic parameter link are the same, as in the example of Case (II), only one cyclic connected data-link list in the longer cyclic parameter link is required.

The case illustrated by the examples in Figure 5.11 is in fact similar to the earlier mentioned case of equal-length cyclic connected data-link lists. The only difference is that several common parameters are involved in the shorter cyclic parameter link in this case while only one common parameter is involved in the previous case. Therefore, if the same value is used in all the involved common parameters, as in Case (II) in Figure 5.11, the method used for the previous case can also be applied to the present case. Further modification is necessary if the values used in these common parameters are different. The presence of different values requires different cyclic connected data-link lists in the longer cyclic parameter link to share the common parameter with the shorter cyclic parameter link in order to avoid common parameter blockage. Therefore, in the case of interdependent cyclic parameter links with lengths in the ratio of an exact multiple and with their cyclic connected data-link lists of equal length, the modification is as follows:

(1) Follow steps (1) and (2) of the modification of the interdependent cyclic parameter link with their lengths and the lengths of their cyclic connected data-link lists in the ratio of an exact multiple.

(2) If the values passing through the common parameters in the shorter cyclic parameter link are all the same, follow step (3) of the above modification too; otherwise, follow step (3) below.

(3) Compare each value in the common parameter value sequence of the cyclic connected data-link lists in the shorter cyclic parameter link with the common parameter value sequence of the cyclic connected data-link lists in the longer cyclic parameter link. If the values are the same, blockage at common parameter is absent from these two cyclic parameter links.

Case (I) in Figure 5.12 is an example which have the lengths of cyclic connected data-link lists not in the ratio of an exact multiple. If only the cyclic connected data-link lists are considered, the ones in the longer cyclic parameter link have length of four while the length of the cyclic connected data-link list in the shorter cyclic parameter link is six. Their ratio is **2:3** which is not an exact multiple. Blockage at common parameter occurs in this example. Its graphical representation in Figure 5.12 can clearly show that blockage occurs at the sixth level of recursion. Because the lengths of the involved cyclic parameter links are in the ratio of an exact multiple, as shown by the graphical representations, the common parameters from the two different cyclic parameter links will be paired up at a fixed interval. The graphical representation in Figure 5.12 shows that the common parameters from the two cyclic parameter links need to be paired up in every four levels of recursion if only two cyclic parameter link from the two groups of cyclic parameter links are considered. (If the two groups of cyclic parameter links instead of two cyclic parameter link out of the two group are considered, the common parameters are paired up in each recursion level.) To avoid blockage at common parameter, the values passing through the point where the common parameters from the two different cyclic parameter link are paired up must be the same. In other words, in the case of interdependent cyclic parameter links with lengths in the ratio of an exact multiple, the same value in the common parameter value sequences from different involved cyclic parameter links should be repeated at the same interval. However, the length of the cyclic connected data-link list also affects the interval of the same value to be repeated in its corresponding common parameter value sequence. As shown by the graphical representation of Case (I), the value *a* or *b* is repeated in every four levels in the cyclic connected data-link list among the parameters *AX/XA*, *BX/XB*, *CX/XC* and *DX/XD* which has the length of four levels while the values *a*, *b* and *c* are repeated in an interval of six recursion levels in a cyclic connected data-link list with the length of six levels. On the one hand, the cyclic connected data-link lists with lengths not in the

ratio of an exact multiple cause the values passing through the common parameter to be repeated in different intervals in different cyclic connected data-link lists. On the other hand, the cyclic parameter links with lengths in the ratio of an exact multiple require the common parameter values to be repeated at the same interval in different involved cyclic connected data-link lists. The conflict is unsolvable. Hence, blockage at common parameter will always happen if the lengths of the different cyclic connected data-link lists of the interdependent cyclic parameter link (which have lengths in the ratio of an exact multiple) are themselves in a ratio that cannot be reduced to an exact multiple.

In Case (II) in Figure 5.12, we present an example which seems to contradict our above conclusion. However, if we examine the cyclic connected data-link lists carefully, we can find that the cyclic connected data-link lists in the first cyclic parameter link among the parameters *AX/XA*, *BX/XB*, *CX/XC* and *DX/XD* does not exactly have the length of four levels as the one in Case (I). With the common parameter values shown between *[]* in the modified notion of cyclic connected data-link list described in Section 5.1.1.2.1, we can clearly see that the cyclic connected data-link list in the first cyclic parameter link which can avoid blockage at common parameter is the cyclic connected data-link list of *11--12--13--14-[b]-11--12--13--14-[a]-11--12--13--14-[c]-11* instead of *11--12--13--14--11*. Although the latter notation only indicates the presence of a cyclic connected data-link list of the length of **4**, the former notation shows the presence of a cyclic connected data-link of the length of **12**. Therefore, the lengths of the cyclic connected data-link lists of the two interdependent cyclic parameter links in Case (II) has a ratio of **12:6** which can be reduced to a ratio of an exact multiple: **2:1**. By examining the two examples in Figure 5.12, we can conclude that the modification for the data analysis method in this case is to modify data analysis slightly to produce cyclic connected data-link lists with common parameter values. If all the cyclic connected data-link lists with common parameter values from different involved cyclic parameter links show their lengths in a ratio of not an exact multiple, it indicates that blockage at common parameter will occur and the corresponding recursive definition can avoid nontermination.

5.1.2 Interdependent Cyclic Parameter Links through Common Subgoals

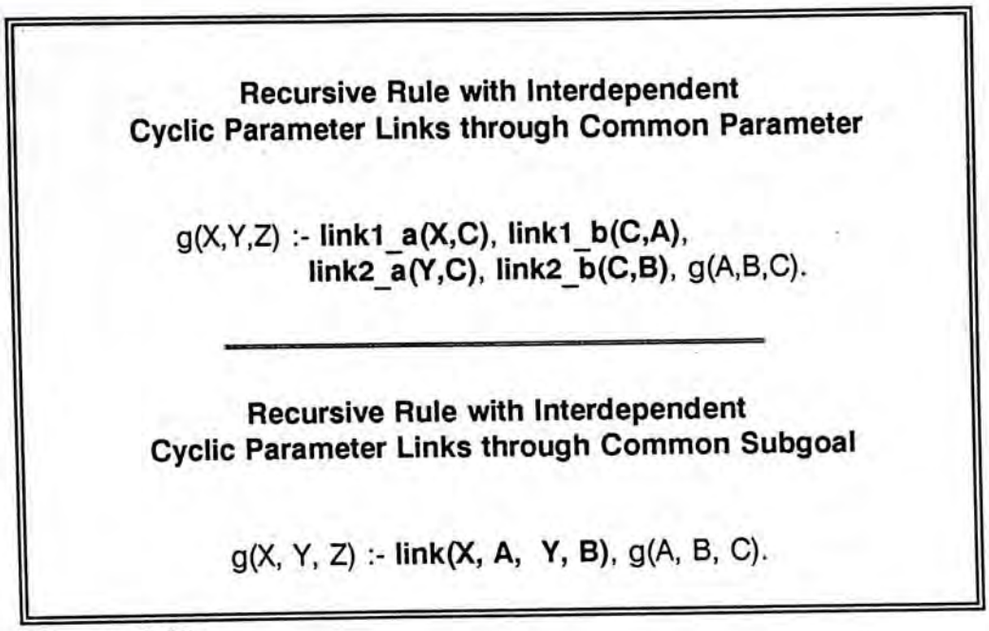


Figure 5.13

In the above discussion, we have seen how interdependency between the different cyclic parameter links in the same recursive definition can be established through some common parameter. We have also seen how interdependent cyclic parameter links can form extra exit conditions that cannot be detected in data analysis and how we can modify data analysis to adapt it to different situations. In this section, we shall examine the another possible way to establish interdependent cyclic parameter links: to establish interdependent cyclic parameter links through common subgoals. At the beginning of Chapter 4, there are two examples of interdependent cyclic parameter links. One of them is a recursive rule with interdependent cyclic parameter links through common parameter while the another is a recursive rule with interdependent cyclic parameter links through common subgoal. These two recursive rules are displayed in Figure 5.13 again. If we just look at the two recursive rules, the two seem to be very different. However, if we express them in graphical representations, the similarity between them can be clearly shown. With a single dotted line to indicate that the two cyclic parameter links are interdependent in Figure 5.14, we can see that all the subgoals involved in both cases must not fail in order to avoid blockage in the infinite data transfer sequences (if any) passing through the cyclic parameter links. To allow the involved subgoals in these interdependent cyclic parameter links not to fail, in the case of common parameter, the values passing through the common parameters in the cyclic connected data-link lists from both involved cyclic parameter links must be the same. On the other hand, **in the case of common subgoals,**

the involved subgoal can only succeed only if the common subgoal is used to form the cyclic connected data-link lists from the two involved cyclic parameter links at the same time. It can be explained by the examples in Figure 5.15.

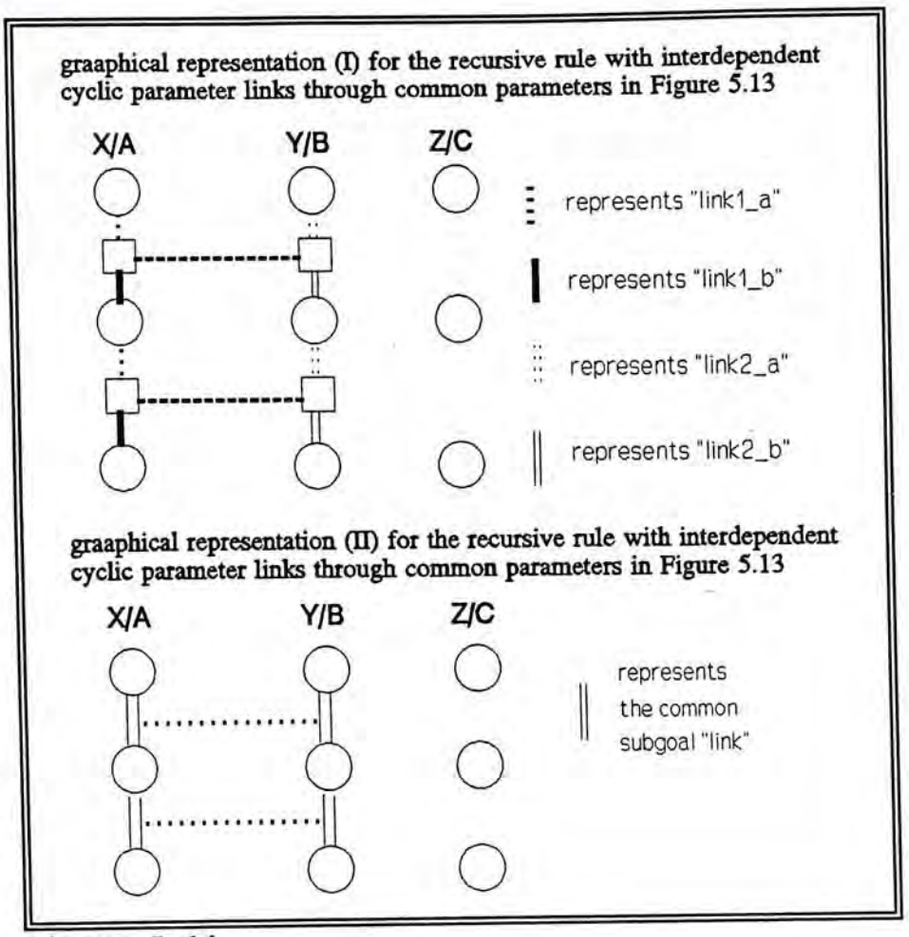


Figure 5.14

The common subgoal in the recursive rule in Figure 5.13 has two different procedures in two different cases in Figure 5.15. In Case (I), as shown by its graphical representation, the recursion is blocked at the fourth level although data analysis can detected a cyclic connected data-link list in each of two interdependent cyclic parameter links. On the one hand, the common subgoal that can allow values to be transferred through the cyclic parameter link between parameters X and A at the fourth level of recursion is *link(2,1,b,c)*. On the other hand, the common subgoal that must be used to pass value through the cyclic parameter link between parameters Y and B at the fourth level of recursion is *link(1,2,a,b)*. Because two different common subgoals are required by the two different involved cyclic parameter links, blockage occurs. In Case (II), the situation is different, with three more facts added to the procedure *link*, the conflict of the subgoals is resolved and nontermination happens as indicated by the graphical representation of Case (II) in Figure 5.15. If we compare the two cases in Figure 5.15 with the examples of interdependent cyclic parameter links through common parameters in Program (b) in Figure 5.1 and Figure 5.2, we immediately recognize the striking similarity between the case of interdependent cyclic parameter links through common parameters and the case of interdependent cyclic parameter links through common subgoals.

They show that the data analysis method fails for the same reason: data analysis cannot recognize blockage at common parameter or common subgoal as a possible exit condition. The reasons for the presence of blockage in both cases are the same: although there are some common parameters or subgoals that are supposed to be shared among the involved cyclic parameter links, the possible data transfer sequences in these cyclic parameter links require conflicting values to pass through some of these parameters or subgoals. The conflict of values can be resolved, as shown in both Case (II) in Figure 5.15 and Figure 5.2, by adding more facts into the corresponding procedures to produce alternate data transfer sequences. Furthermore, when the multi-level interdependent cyclic parameter links are considered, the similarity between the interdependent cyclic parameter links through common parameters and the interdependent cyclic parameter links through common subgoals is evident in the similarity of the relations of the lengths of the interdependent cyclic parameter links and the relations of the lengths of the involved cyclic connected data-link lists. The examples and their graphical representations in Figure 5.16 show that the three types of relations of the lengths of interdependent cyclic parameter links are also present in the interdependent cyclic parameter link through common subgoals. Similarly, the cyclic connected data-link lists in the interdependent cyclic parameter links through common subgoals can be related together in terms of their lengths as the cyclic connected data-link lists in the case of

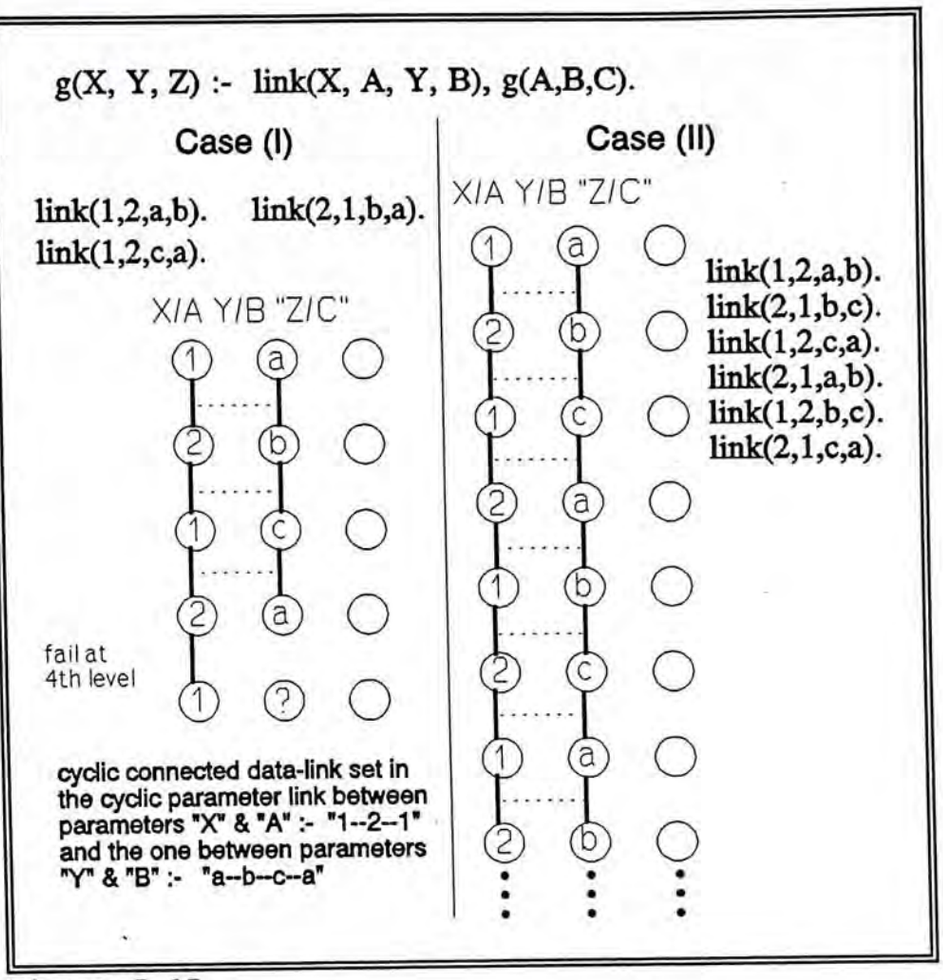
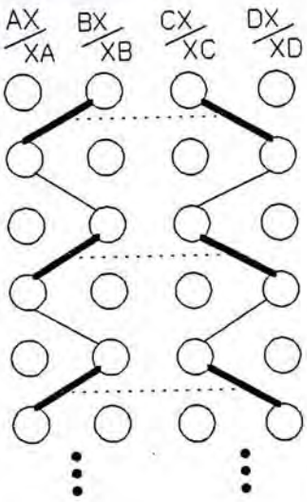


Figure 5.15

Case (I)

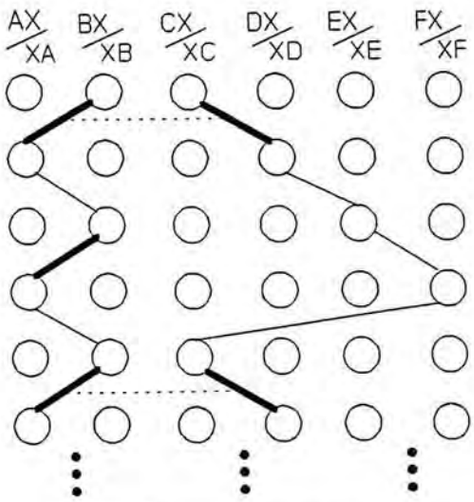
```
goal(AX,BX,CX,DX) :-
    link1(AX,XB),
    link2(DX,XC),
    clink(BX,XA,CX,XD)
    goal(XA,XB,XC,XD).
```

the example of inter-dependent cyclic parameter link with equal length



Case (II) the example of interdependent cyclic parameter links with their lengths in a ratio of an exact multiple

```
goal(AX,BX,CX,DX,EX,FX) :-
    link1(AX,XB), link2b(DX,XE), link2c(EX,XF),
    link2d(FX,CX), clink(BX,XA,CX,XD),
    goal(XA,XB,XC,XD,XE,XF).
```



Case (III) the example of interdependent cyclic parameter links established through subgoals with thier lengths not in a ratio of an exact multiple

```
goal(AX,BX,CX,DX,EX) :- link1(AX,XB), link2b(DX,XE), link2c(EX,CX),
                        clink(BX,XA,CX,XD), goal(XA,XB,XC,XD,XE).
```

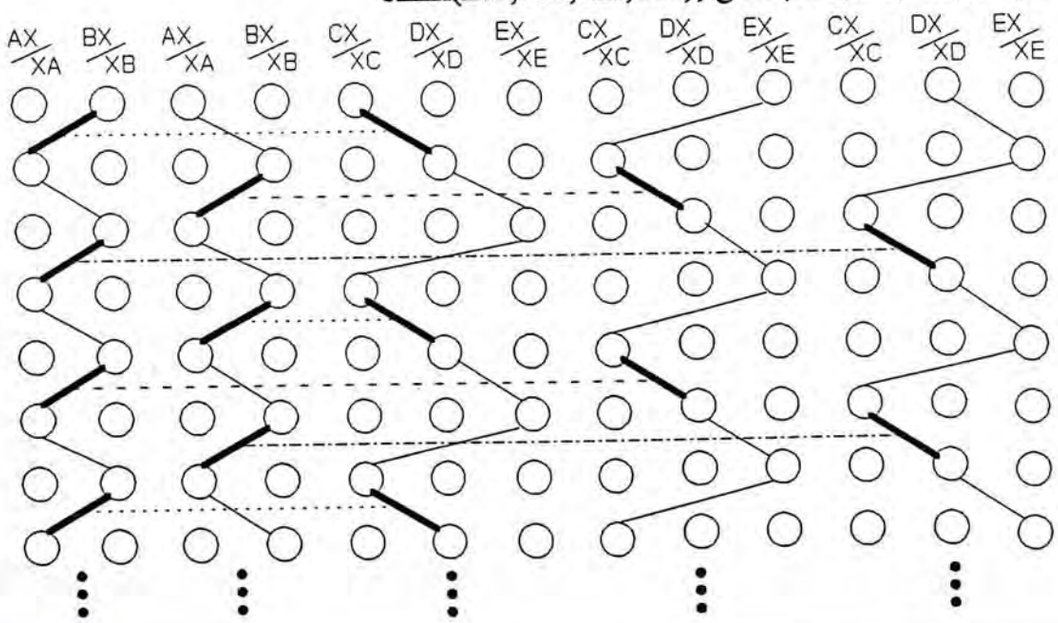


Figure 5.16

common parameter. In Figure 5.17, we show some examples of the case of equal-length cyclic connected data-link lists, the case of cyclic connected data-link lists with lengths

in the ratio of an exact multiple, and the case of cyclic connected data-link lists with lengths not in the ratio of an exact multiple in a recursive definition with interdependent cyclic parameter links of equal length. All these similarities between the interdependent

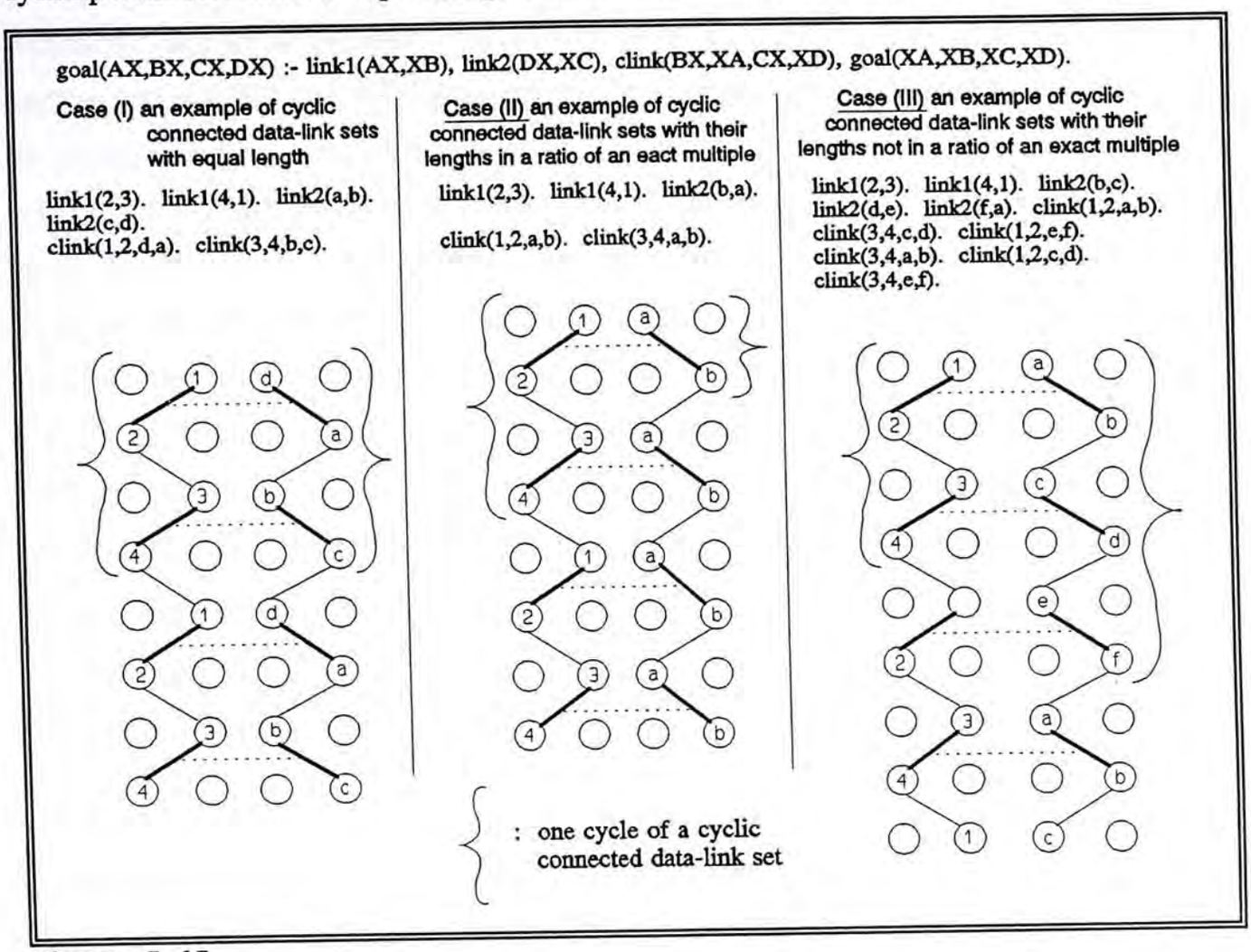


Figure 5.17

cyclic parameter links through common parameters and the interdependent cyclic parameter links through common subgoals show that the above discussion on how to adapt data analysis to handle the presence of interdependent cyclic parameter links through common parameters can also be applied to the presence of interdependent cyclic parameter links through common subgoals with only slight modifications.

In the case of common parameters, we can detect blockage at common parameter by comparing the common parameter value sequences from different cyclic parameter links. The absence of blockage at common parameter is indicated by the presence of a common parameter value sequence shared among all involved cyclic parameter links. However, in the case of common subgoals, there are no common parameter value

sequences to be compared. We need another indicator. **The indicator is the common subgoal sequence**. Data analysis can be modified to record each common subgoal used during the cyclic connected data-link lists construction to form a sequence which is the common subgoal sequence. In Figure 5.17, the cyclic parameter link between parameters *AX/XA* and *BX/XB* forms the cyclic connected data-link list of *2--3--4--1--2* in all three cases. Its corresponding common subgoals sequence is *clink(3,4,b,c)--clink(1,2,d,a)* in all three cases as well. Moreover, we can also modify the notion of cyclic connected data-link list in a way similar to the case of common parameters to incorporate the value of the common subgoals used in a particular cyclic connected data-link list. For example, the cyclic connected data-link list of *2--3--4--1--2* can be transformed to *2--3-[clink(3,4,b,c)]-4--1-[clink(1,2,d,a)]-2*. The value between *[]* indicates the common subgoal responsible for forming the particular segment of the cyclic connected data-link list. There is no blockage at common subgoal between two cyclic parameter links if their common subgoal sequences are the same. For example, for Case (I) in Figure 5.15, if data analysis is used, the cyclic connected data-link list in the cyclic parameter link between parameters *X* and *A* is *1--2--1*. But this cyclic connected data-link list can have two different common subgoals sequences: *link(1,2,a,b)* or *link(1,2,c,a)*. For the cyclic parameter link between parameters *Y* and *B*, the cyclic connected data-link list is *a--b--c--a* and there is only one possible common subgoal sequence which is *link(1,2,a,b)--link(2,1,b,c)--link(1,2,c,a)*. Obviously, the common subgoals sequences from these two cyclic parameter links are different. The conclusion drawn from this difference coincides with the result shown by the graphical representation. On the other hand, in Case (II) in Figure 5.15, if common subgoals are also considered in constructing the connected data-link list. One of the cyclic connected data-link lists that can be formed in the first cyclic parameter link between parameters *X* and *A* is *1-[link(1,2,a,b)]-2-[link(2,1,b,c)]-1-[link(1,2,c,a)]-2-[link(2,1,a, b)]-1-[link(1,2,b,c)]-2-[link(2,1,c,a)]-1*. The common subgoal sequence therefore is *link(1,2,a,b)--link(2,1,b,c)--link(1,2, c,a)--link(2,1,a,b)--link(1,2,b,c)--link(2,1,c,a)*. For the cyclic parameter link between parameters *Y* and *B*, we can also form an equal-length cyclic connected data-link list as *a-[link(1,2,a,b)]-b-[link(2,1,b,c)]-c-[link(1,2,c,a)]-a-[link(2,1,a,b)]-b-[link(1,2,b,c)]-c-[link(2,1,c,a)]-a*. The common subgoal sequence therefore is also *link(1,2,a,b)--link(2,1,b,c)--link(1,2,c,a)--link(2,1,a,b)--link(1,2,b,c)--*

link(2,1,c,a). So the same common subgoal sequence in the two interdependent cyclic parameter links indicates that there is no blockage at common subgoal between these two interdependent cyclic parameter links. The graphical representation of Case (II) in Figure 5.15 can confirm this conclusion. Therefore, by replacing the step of comparing common parameter value sequence with the **step of comparing common subgoal sequence**, the modification of data analysis for the case of common parameters can be adapted to the case of common subgoals.

5.1.3 Interdependent Cyclic Parameter Links with Special Parameters

So far, our discussion has included only the cases of interdependent cyclic parameter links formed by subgoals only. In this section, we shall look at interdependent cyclic parameter links formed by special parameters. To reduce redundancy, our discussion shall focus on interdependent cyclic parameter links formed by lists because the similarity between lists and structured data, and lists are more common in Prolog programming.

Although there are two kinds of interdependent cyclic parameter links for the cyclic parameter links formed by subgoals, i.e, the interdependent cyclic parameter links through common parameters or through common subgoals, there is only one type of interdependent cyclic parameter links for the cyclic parameter links formed by special parameter only. The reason is obvious. Since the cyclic parameter links formed by special parameters contain no subgoal, interdependency must be established among the cyclic parameter links through some common parameters. Of course, we can establish interdependent cyclic parameter links through common subgoals in the case of cyclic parameter links formed by both special parameters and subgoals. However, this is a rare case and the case of interdependent cyclic parameter links with both special parameters and subgoals is even more rare. Therefore, we do not go into detail. The general principle for detecting any blockage at common subgoal in the case of the interdependent cyclic parameter links formed by both special parameters and subgoals is similar to the case of the interdependent cyclic parameter links formed by subgoals.

As shown in Section 4.5.2, we can construct connected data-link lists in the cyclic parameter link with both special parameters and subgoals in a way similar to the case of cyclic parameter links with subgoals. Therefore, we can detect the infinite data transfer sequence in cyclic parameter links with both subgoals and special parameters by detecting the presence of cyclic connected data-link lists. If cyclic connected data-link lists can be constructed in the interdependent cyclic parameter links with both subgoals and special parameters and the interdependency is established through the common subgoal, the modification for data analysis in the case of the interdependent cyclic parameter links formed by only subgoals can also be applied to the case of the interdependent cyclic parameter links formed by both subgoals and special parameters. The discussion in Section 5.1.2 can be applied to this case without any adjustment.

Before we discuss how interdependent cyclic parameter links through common parameters can be formed by the cyclic parameter links with special parameters only, we shall first examine a special case. The well-known recursive definition *append* below can be an example of the special case. In the recursive definition *append*, there are

```
append([E|X], Y, [E|Z]) :- append(X, Y, Z).

append([], L, L).
```

two cyclic parameter links. At first glance, they are interdependent because the parameter E is shared between them. However, if we examine the concept of cyclic parameter link, we shall immediately notice that **the parameter E actually is not responsible for transferring data into the next level of recursion.** It is the tails of the lists in both cyclic parameter links, X and Z, that are passed into the next level. The two cyclic parameter links are interdependent because blockage at common parameter can happen if the values in the common parameter E are different in these two involved cyclic parameter links. Therefore, the common parameter E can be an exit condition in some situations. This will not be obvious if the recursive definition *append* is considered alone. However, the common parameter E can be clearly shown as an exit condition by

```
is_prefix(Prefix, List) :-
        append(Prefix, Remain, List).

append([E|X], Y, [E|Z]) :- append(X, Y, Z).
append([], L, L).
```

using *append* to build the definition *is_prefix*: in the definition *is_prefix*, the recursive definition *append* will terminate at the point where the list in the parameter *Prefix* becomes empty or **the first element in *Prefix* is different from the first element in *List*.** If the recursive definition *append* is considered, the recursion stops when the values passing through the common parameter E in the two interdependent cyclic parameter links are different. Hence, the common parameter acts as an exit condition in this situation.

However, if data analysis is applied to the recursive definition *append*, it will detect no infinite data transfer sequences in both interdependent cyclic parameter links. When the evaluation of the recursive definition goes one level further, the lengths of the connected data-link lists in both cyclic parameter links are one element shorter. Hence, data analysis will yield the conclusion that the data transfer sequences can only be finite in both cyclic parameter links. However, in the case of common parameters, data analysis can correctly predict the result without any concern for the extra exit condition provided by the common parameter.

Therefore, for the interdependent cyclic parameter links formed by special parameters, the data analysis method only needs to be modified to handle those which can form infinite data transfer sequences. As shown in Chapter 4, in the case of special parameters, an infinite data transfer sequence implies that the lengths of the data passing through all the involved cyclic parameter links either increase or remain unchanged during the evaluation of the recursive definition. Therefore, we can change the recursive definition *append* into the following recursive definition, which has interdependent cyclic parameter links through common parameters and requires consideration on its common parameter:

```
apd(X, Y, Z) :- apd([E|X], Y, [E|Z]).

apd([], L, L).
```

In this recursive definition *apd*, the data transfer sequences in both interdependent cyclic parameter links are infinite as indicated by some connected data-link lists with growing lengths. Although a common parameter E appears in both cyclic parameter links, the common parameter cannot act as an exit condition in this recursive definition. As mentioned in our discussion on the case of interdependent cyclic parameter links formed by only subgoals shown, the mere presence of common parameters does not necessarily lead to blockage at common parameter. Nontermination can be avoided by common parameter blockage only when the values passing through the common parameter in different cyclic parameter links are in conflict. However, in order to have any conflict in values, at least some values must be transferred through the common parameter during the recursion. But if we examine the recursive definition *apd* again, we can see that no value is transferred through the common parameter E during the recursion. In fact, the common parameter E remains uninstantiated during the infinite evaluation.

By the example of the recursive definition *apd*, we can understand that the common parameter in the interdependent cyclic parameter links formed by only special parameters cannot provide extra exit conditions as in the ones formed by subgoals only. Without any impurities, there is no input/output predicate in a pure Prolog program to assign value to the parameter during the evaluation of a recursive definition. However, an infinite data transfer sequence in the interdependent cyclic parameter links formed by special parameters can only occur when the data passing through the cyclic parameter links are lists of increasing length or lists of fixed length. **In the case of increasing length,** new elements must be added into the list during the recursion. But since there is no other way to input new values into a recursive definition during its evaluation except through the arguments of the recursive definition themselves, **the new element that can be added into the list during the recursion can only be an uninstantiated variable** as shown by the recursive definition *apd*. Uninstantiated common parameters

do not have any conflict in values and no blockage at common parameter will occur. If the infinite data transfer sequence is formed by passing some lists of fixed length during the recursion, the values passing through the common parameter either **are always some new values, appearing in a repeating sequence** or **just are some uninstantiated variables.** As stated above, the values passing through the common parameter cannot always be new values, therefore, the values passing through the common parameter can either form a repeating sequence or are made up by some uninstantiated variables. If only uninstantiated variables appear in the common parameter, blockage at common parameter will never occur. Then the only remaining possibility of common parameter blockage is the case of common parameters with repeating values. The above recursive definition *apd* is altered to provide the example below.

```
appd([E|X], Y, [E|Z]) :- appd([E|X], Y, [E|Z]).
appd([], L, L).
```

In *appd*, if any recursion can occur, the lists passing through both interdependent cyclic parameter links will remain the same and the values passing through the common parameter E will repeat infinitely. Although we can have the common parameter E to be instantiated with some value in this case, the values passing through E will not be different in different cyclic parameter links. On the other hand, if there is a conflict of values in the argument of the recursive definition, eg., appd([1,2,3],Y,[4,5,6]), no recursion will ever happen. So, either there is no recursion or no blockage is possible in this situation. In conclusion, we find that the common parameter cannot an extra exit condition once infinite data transfer sequences occur. If we analyze the reason for the absence of blockage at common parameter, we can realize that **the infinite data transfer sequences in the cyclic parameter links formed by special parameters can appear only when there is no blockage at common parameter possible.** Therefore, no modification is needed for the data analysis method when the interdependent cyclic parameter links with only special parameters are considered.

Before we close our discussion on the interdependent cyclic parameter links with special parameters, we must consider the possibility of forming **interdependent cyclic parameter links between the cyclic parameter links formed by special parameters alone and the cyclic parameter links formed by subgoals only**. Although it is a rare case to mix these two types of cyclic parameter links together, we would like to examine this case to complete our discussion on the interdependent cyclic parameter links with special parameters. Let us consider the example below:

```
goal(X, Y, Z) :- linka(Y,E), linkb(E,B),
                 goal([E|X], B, [E|Z]).
goal([], L, L).

      linka(1,a).    linkb(a,2).
      linka(2,b).    linkb(b,1).
```

In the recursive definition *goal*, there are three interdependent cyclic parameter links. Two are formed by special parameters and one by subgoals. The common parameter *E* is shared by all three. Although the uninstantiated variables no longer appear in the common parameter, no common parameter blockage is possible in this example. In order to have any conflict of values at the common parameter, the cyclic parameter links formed by the special parameters need to have some solid values to be passed along. However, as mentioned in the above discussion, this is not possible if any infinite data transfer sequence can be formed in the cyclic parameter links with only special parameters. Therefore, there is no need to modify the data analysis method in the case of interdependent cyclic parameter links that are formed from cyclic parameter links with only special parameters and cyclic parameter links with only subgoals.

5.2 A Special Case of Cyclic Parameter Links established through Special Parameters

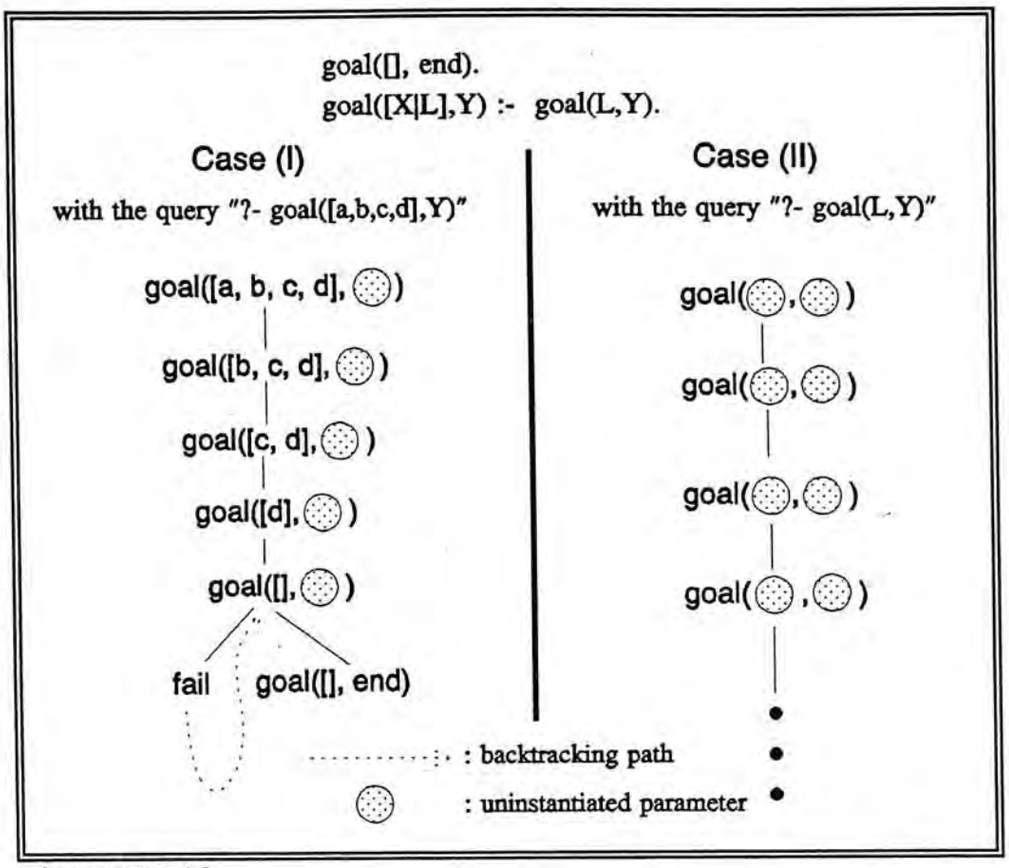


Figure 5.18

In the previous section, we have discussed the situation in which data analysis may overlook some potential exit conditions and give a false warning of nontermination. In this section, we shall examine the situation in which data analysis fails to indicate possible nontermination. In Figure 5.18, there is a simple recursive definition with only one cyclic parameter link which is formed by special parameters. Because the length of the list passed into the next level of recursion through the parameter L is one element shorter than the list in the previous level, data analysis will conclude that the data transfer sequence in the cyclic parameter link is finite and will consider the corresponding recursive definition to be free of nontermination. However, the search trees in Figure 5.18 show the above conclusion to be only partially true. In general, the evaluation of a recursive definition can be terminated properly if any list (or any value) is supplied to be the first argument of the query. As shown by the search tree of Case (I), the list supplied is shortened by one element in each level of recursion and eventually brings the recursion to an end when the list becomes empty. But Case (II) shows that the evaluation of the recursive definition in Figure 5.18 can end in nontermination in a special situation: i.e., **when an uninstantiated variable is supplied to the cyclic parameter link.** The search tree of Case (II) in the figure shows how

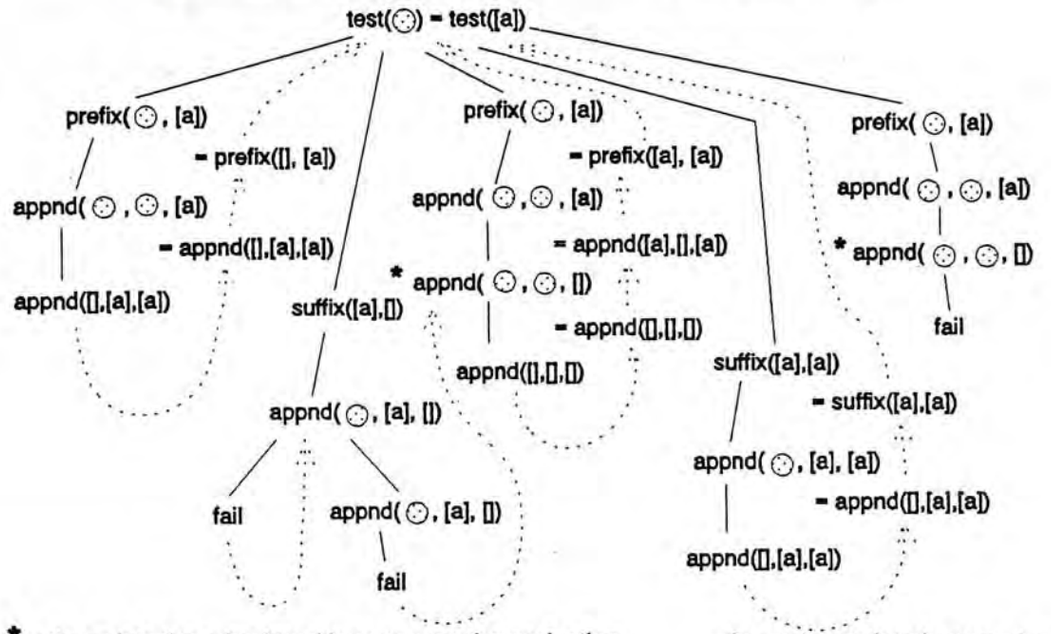
nontermination occurs. Unlike an empty list or a value, an uninstantiated variable can succeed to be instantiated with the list *[X|L]*. So the *head-tail separator*, |, cannot act as an exit condition. This is true for the operator , as well. Usually, such a problem is considered as part of the well-known *occur check* problem.

In other words, for the cyclic parameter link established through special parameters, we can be certain about the presence or absence of nontermination only after we know what data are supplied to the cyclic parameter link when the recursive definition is evaluated. However, there is no easy solution to this problem in an analytical approach. The data supplied to the cyclic parameter link established through special parameters are usually unknown until it is evaluated. For example, there is no way to tell whether the query *?- goal([a,b,c,d], Y)* or the query *?- goal([], Y)* will be used if we just analyze the recursive definition *goal*. In more complicated Prolog programs, the situation can become even worse. Merely analyzing the recursive definition or even the entire program cannot determine what data will be supplied to a particular cyclic parameter link in a particular recursive definition. That can only be done by checking the supplied data **during the evaluation**.

Figure 5.19 shows how the situation becomes more complicated if more than one recursive definition are involved in the same recursive rule while a variable is supplied to the cyclic parameter link with special parameters in one recursive definition. There are two semantically identical Prolog programs in Figure 5.19. The only difference between them is the position of the definitions *prefix* and *suffix*. In Program (a), the definition *prefix* comes first while in Program (b) the definition *suffix* comes first. In both definitions, there is a recursive definition *appnd*. If it is analyzed with the data analysis method, the cyclic parameter link in *appnd* is established through special parameters and the data transfer sequence is finite because the list passed into the next level of recursion is shorter than the one from the previous level. **Therefore, according to data analysis, no nontermination will occur in both programs.** However, the search trees in Figure 5.19 contradict this conclusion. Because the variable *L* is supplied to the

Program (a) test(L) :- prefix(L,[a]), suffix([a],L).
prefix(X,L) :- appnd(X,Y,L). suffix(Y,L) :- appnd(X,Y,L).
appnd([],L,L). appnd([H|X],Y,[H|Z]) :- appnd(X,Y,Z).

Search Tree (a) using the query "?- test(L)" for Program (a)



Program (b) test(L) :- suffix([a],L), prefix(L,[a]).
prefix(X,L) :- appnd(X,Y,L). suffix(Y,L) :- appnd(X,Y,L).
appnd([],L,L). appnd([H|X],Y,[H|Z]) :- appnd(X,Y,Z).

Search Tree (b) using the query "?- test(L)" for Program (b)

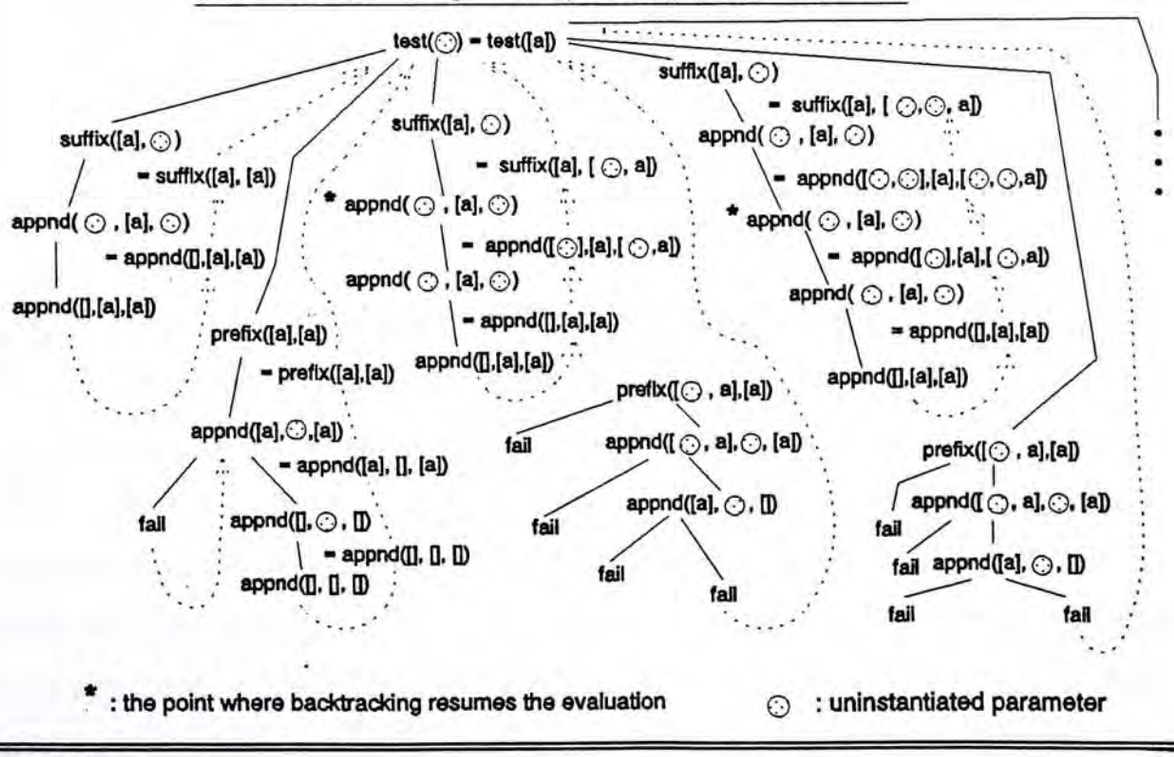


Figure 5.19

second parameter of the subgoal *suffix* in both programs, according to the procedure of *suffix*, variables are then supplied to the first and third parameter of the subgoal *appnd*. If we examine the procedure of *appnd*, we can see both the first and third parameter are the special parameters used in forming the two cyclic parameter links. This implies that variables are indeed supplied to the cyclic parameter links by the subgoal *suffix([a], L)* in both programs in Figure 5.19. **According to our earlier discussion in this section, we can conclude that nontermination will occur in both programs.**

But the search trees in Figure 5.19 again contradict this conclusion to be wrong. Search tree (b) for Program (b) shows that nontermination will occur in Program (b) and the reason of nontermination is exactly what we have discussed above: a variable is supplied to the cyclic parameter link established through special parameters. However, Search tree (a) indicates that it is difficult to approach the problem by merely analyzing the program. Search tree (a) reveals why Program (a) can terminate. The variable *L* can be instantiated to a certain value after the evaluation of the subgoal *prefix*. Because the subgoal *suffix([a], L)* is placed after the subgoal *prefix(L, [a])*, the parameter *L* in the subgoal *suffix* is no longer an instantiated parameter. Search tree (a) shows either list *[]* or *[a]* will be instantiated with *L* when the subgoal *suffix* is evaluated. Because the content of a parameter changes during the evaluation of a program, **the real value that is supplied to a cyclic parameter link during the evaluation of a recursive definition can hardly be predetermined by an analytical approach alone.** However, one can handle this situation by tracing the input data for the cyclic parameter link established with only special parameters during the recursion.

Although the data analysis method cannot detect nontermination caused by the inappropriate data supplied, **it can help to discern its cause.** After data analysis yields the conclusion that a cyclic parameter link with only special parameter can only have a finite data transfer sequence, our discussion in this section indicates that any infinite data transfer sequence appearing in this cyclic parameter link must have resulted from the supply of inappropriate data. This piece of information will be very helpful to a Prolog programmer remedying the problem. Since nontermination arises when any uninstantiated variable is supplied to the cyclic parameter link established through

special parameters, nontermination due to this cause can be easily detected in a run-time tracing approach by detecting any attempt to pass an uninstantiated variable to a cyclic parameter link. Although it is hard to develop an algorithm to handle the entire *occur check* problem, it can be seen that the present problem is much less serious and can be tackled with a tracing approach.

CHAPTER 6 —Results and Conclusion

In the foregoing chapters, the cause of nontermination in Prolog programs was analyzed, algorithms were developed, and special recursive programs were investigated. In this concluding chapter, the results of the study are summarized; the limitations of the algorithms are explained; some future research works are suggested, and a conclusion is finally presented.

6.1 The Results and Implications

This thesis develops a new approach to detect nontermination errors in pure Prolog programs, providing a theoretical framework based on static program structure analysis for nontermination detection. The new method for nontermination detection involves essentially the following steps: *program structure analysis*, *parameter analysis* and *data analysis*.

Through program structure analysis, all the recursive rules in a given Prolog program are identified. Then for each recursive procedure in a given program, parameter analysis is carried out on each of its recursive rules. If no cyclic parameter links can be found, the recursive procedure would not be able to terminate. According to our findings, such nontermination can be attributed to an improper definition of the recursive procedure. The absence of a cyclic parameter link implies that it is impossible to pass parameter values from the initial procedure call into successive parameter cycles or levels of recursion. As an exit condition should contain one or more variables related to the formal parameters of a recursive procedure, if the exit variables are not related to the formal parameters, the underlying exit condition cannot be a proper one.

In the absence of a cyclic parameter link, there cannot be a proper exit condition for the recursive procedure. Therefore, nontermination will occur.

On the other hand, the presence of a cyclic parameter link in recursive procedure implies that the parameter values first input from a procedure call to the recursive procedure can be modified and transferred to successive parameter cycles or levels of recursion. Therefore, the presence of a cyclic parameter link also implies that there is a parameter modifying mechanism also known as *parameter modifying process* which is usually a conjunction of one or more subgoals preceding the recursive subgoal, which modifies the values of the formal parameters to other values for passing to the recursive subgoal. Such a parameter modifying process is also a potential exit-reaching process since there is a possibility that the initial parameter values may be so modified during successive parameter cycles of recursion that they may eventually reach the state required to satisfy the exit condition. Consequently, with a cyclic parameter link alone, we can only conclude that there is a potential exit-reaching process. However, we cannot be certain whether the potential exit-reaching process is a real exit-reaching process; that is, whether the potential exit-reaching process can modify the parameter values of the recursive procedure to reach the exit condition during recursion. In order to confirm whether the recursive procedure can terminate, data analysis has to be carried out.

For each cyclic parameter link, the data analysis algorithm first attempts to construct a set of data links. Each data link represents the presence of certain parameter (or data) values passing through the corresponding cyclic parameter link for at least one parameter cycle of recursion. Then the data links are connected together as far as possible, forming a set of connected data-link lists. Each finite connected data-link list represents certain parameter values being able to be transferred over the corresponding cyclic parameter link through several parameter cycles of recursion. From the data transfer analogy, a connected data-link list represents a data transfer sequence over the cyclic parameter link. If there is at least one cyclic data-link list, the recursion of the procedure will not terminate; a cyclic data-link list implies that all or part of the underlying parameter values can be repeatedly transferred over the cyclic

parameter link during recursion. So, a cyclic data-link list corresponds to an infinite data transfer sequence. In this situation, we can confirm that the parameter modifying process is not an exit-reaching process since it cannot modify during recursion the parameter values to satisfy the condition for exiting the recursion. On the other hand, if the set of connected data-link lists contains only finite connected data-link lists, the recursive procedure can be confirmed to have no nontermination problem. It can also be confirmed that parameter modifying process is an exit-reaching process.

For a cyclic parameter link established through special parameters such as those based on list structures, instead of constructing a set of connected data-link lists, the data analysis algorithm attempts to find out the *change tendency* of the data (or parameter) values over the cyclic parameter link for one parameter cycle of recursion. If *change tendency* is not a negative value, the recursive program cannot terminate as a negative *change tendency* value represents the presence of an infinite data transfer sequence over the cyclic parameter link.

6.2 Limitations and Future Research

Although parameter analysis can generally be used to detect the presence of any potential exit-reaching process by constructing cyclic parameter links, data analysis is limited to pure Prolog programs. As pure Prolog is a subset of general Prolog, data analysis cannot be always applied to general Prolog programs. However, further research should be conducted for incorporating the data analysis and certain run-time tracing techniques for the detection of nontermination in general Prolog programs. Moreover, our compile-time approach to nontermination detection provides a sound foundation for the implementation of a relatively powerful nontermination diagnosis system.

Another limitation of our present data analysis algorithm is that no solution has been found for nontermination caused by the *input parameter* problem; in a Prolog

recursive procedure, some of its special parameters should be classified as input parameters. That is, when such a procedure is called, those input parameters should be instantiated to some values. For instance, the data analysis algorithm will conclude that there is no nontermination problem in the recursive definition defined by the recursive rule below:

append([X|L], L_, [X|L_L]) :- append(L, L_, L_L).

It is obvious that the above recursive rule will terminate only if the first parameter of append is bound to a ground list structure when it is invoked. Otherwise, a query of the form ?-append(X,Y,Z) can cause the evaluation of this recursive definition to become nonterminating. In fact, Plümer has worked on this problem with a mathematical approach [9]. Evidently, a compile-time method cannot easily handle the problem since the termination of this type of procedure will depend on whether there is any ground value bound to its input parameter.

One more limitation of the present data analysis algorithm is that it cannot handle the situation of having multiple recursive subgoals in a single recursive rule. Although we have illustrated in Figure 4.16 in Chapter 4 how data analysis can handle the case of multiple recursive subgoals in a single recursive rule, there is a subset of the case which data analysis cannot deal with. As an example, consider the recursive rule below:

goal(X,Y) :- subgoal_1(X,A), goal(A,B), goal(B,C).

Although our data analysis can deal with the first recursive subgoal in this recursive rule, it cannot handle the second one because the cyclic parameter link for the second recursive subgoal is established through the first recursive subgoal. Though the present algorithm can determine whether the first recursive subgoal in the recursive rule can terminate or not, it cannot find out what data can be passed in and out of the parameters A and B. Therefore, it cannot be applied to the second recursive definition. The two examples in Section 5.2 of Chapter 5 also demonstrate that the present algorithm cannot handle the case of a cyclic parameter link established through special

parameters. Since a recursive rule with multiple recursive subgoals is quite important in Prolog programming, further research should be done in this direction.

Chapter 5 shows that the data analysis algorithm formulated in Chapter 4 has another limitation. That is, it does not handle a recursive definition with interdependent cyclic parameter links. In fact, the interdependency between different cyclic parameter links can establish a kind of exit condition in some situations. Chapter 5 has already provided some useful ideas concerning how to handle this type of recursive definitions. Therefore, what remains to be done is to spend more effort on reformulating the present data analysis algorithm in order to include the ideas depicted in Chapter 5.

The present parameter analysis and data analysis algorithms focus only on sequential Prolog programs. It may be fruitful to do more research to explore using and expanding the techniques devised in thesis to handle the nontermination problems in parallel Prolog programs.

6.3 Conclusion

This thesis develops a compile-time analytical approach for detecting nontermination in pure Prolog programs. The underlying theories and algorithms can serve as a sound framework for nontermination detection.

Before the algorithms in Chapters 3 and 4 were formulated, the cause of nontermination of pure Prolog programs was first examined. In the context of a Prolog recursive definition, an exit condition should contain variables related to the formal parameters. The exit-reaching process, which modifies the exit condition variables to reach the state for exiting the recursion, is in general a conjunction of one or more subgoals preceding the recursive subgoal in a recursive rule. In fact, reaching an exit condition means that reaching the state at which the conjunction of subgoals

representing the exit condition fails, thus preventing any further recursion from taking place. Therefore, our method of nontermination detection involves detecting the presence of an exit condition and the necessary exit-reaching process.

The parameter analysis algorithm presented in Chapter 3 can be used to analyze a recursive rule to check for the presence of an exit condition by constructing cyclic parameter links. The absence of any cyclic parameter link may be interpreted as having an exit condition; thus the recursive definition will not terminate. It also means that the recursive rule has not been properly defined. On the other hand, the presence of one or more cyclic parameter links for a recursive definition means that there is an exit condition, implying there is a parameter modifying process which is a potential exit-reaching process. To confirm the presence of an exit-reaching process, data analysis has to be performed. Subsequently, it can be concluded whether the recursive procedure can terminate.

Though restricted to handling pure Prolog programs, the present algorithms are relatively powerful when compared with those devised by Shapiro. They can work quite independently. In general, they do not require the user to supply any information, whereas Shapiro's nontermination diagnosis algorithm requires the user to provide a query and a stack depth. The result of his algorithm may rely on the information provided the user. It is believed that the present data analysis algorithm can be enhanced considerably to cope the special types of recursive definitions. However, the parameter analysis algorithm is applicable to both pure and general Prolog programs. Further, the compile-time analytical techniques developed in this thesis and the run-time tracing techniques used by Shapiro can be merged together to formulate more powerful nontermination diagnosis algorithms for general Prolog programs. Lastly, the parameter link graphs and the data link graphs developed in this thesis have shown to be powerful for facilitating respectively the analysis of parameters and data transfer in a recursive procedure.

Reference

[1] M.A. Covington. **Eliminating Unwanted Loops in Prolog,** *ACM SIGPLAN Notices*, 1985, Vol. 20, #1.

[2] M.A. Covington. **Further Note on Looping in Prolog,** *ACM SIGPLAN Notices*, 1985, Vol. 20, #8.

[3] M. Baudinet. **Proving Termination Properties of PROLOG Programs: A Semantic Approach,** *the Proc. of the Symposium on Logic in Computer Science, IEEE*, July 1988.

[4] D.R. Brough, C.J. Hogger. **The Treatment of Loops in Logic Programming,** *Technical Report DoC 86/16, Department of Computing, Imperial College of Science and Technology*, London, September 1986.

[5] D.R. Brough, A. Walker. **Some practical properties of logic programming interpreters,** *Proc. FGCS*, 1984, p.149-156.

[6] A. Van Gelder. **Efficient Loop Dectection in Prolog--Using the Tortoise-and-Hare Technique,** *J. Logic Programming*, 1987, #4, p.23-31.

[7] R. Kowalski. **Logic for Problem Solving,** *North Holland*, 1979.

[8] D. Nute. **A Programming Solution to Certain Problems with Loops in Prolog,** *ACM SIGPALN Notices*, 1985, Vol. 20, #8.

[9] L. Plümer. **Termination Proofs for Logic Programs based on Predicate Inequalities.**

[10] D. Poole, R. Goebel. **On Eliminating Loops in Prolog,** *SIGPLAN Notices*, 1984, Vol. 20, #8, p.38-40.

[11] D. De Schreye, K. Verschaetse, M. Bruynooghe, **On the Existence of Nonterminating Queries for a Restricted Class of Prolog-clauses,** *Artificial Intelligence*, 41, 1989, p.237-248

[12] D. De Schreye, K. Verschaetse, M. Bruynooghe, **A practical technique for detecting non-terminating queries for a restricted class of Horn clauses, using directed, weighted graphs.**

[13] E.Y. Shapiro. **The Art of Prolog,** *MIT Press*, 1986.

[14] E.Y. Shapiro. **Algorithmic Program Debugging,** *MIT Press*, 1983.

[15] *Ibid.*, p.59.

[16] J.D. Ullman, A. Van Gelder. **Efficient tests for top-down termination of logical rules,** *JACM 35*, (1988), p.345-373.

[17] W.F. Clocksin, C.S. Mellish. **Programming in Prolog,** *Springer-Verlag*, 1981