Technical Report 904

Automated Program Recognition

Linda M. Wills

MIT Artificial Intelligence Laboratory

Automated Program Recognition

by

Linda Mary Wills[1]

Massachusetts Institute of Technology

February 1987

Revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on May 16, 1986 in partial fullfillment of the requirements for the degree of Master of Science.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124 and in part by the National Science Foundation Grant No. DCR-8117633.

[1] Formerly known as Linda M. Zelinka

Abstract

The key to understanding a program is recognizing familiar algorithmic fragments and data structures in it. Automating this recognition process will make it easier to perform many tasks which require program understanding, e.g., maintenance, modification, and debugging. This report describes a recognition system, called the Recognizer, which automatically identifies occurrences of stereotyped computational fragments and data structures in programs. The Recognizer is able to identify these familiar fragments and structures, even though they may be expressed in a wide range of syntactic forms. It does so systematically and efficiently by using a parsing technique. Two important advances have made this possible. The first is a language-independent graphical representation for programs and programming structures which canonicalizes many syntactic features of programs. The second is an efficient graph parsing algorithm.

Thesis Supervisor: Dr. Charles Rich
Title: Principal Research Scientist

Acknowledgments

I am grateful to Chuck Rich, my thesis advisor, for generously sharing his ideas and advice on both technical issues and personal decisions. His enthusiasm and support make working with him very enjoyable. Dan Brotsky took some of the early steps in this area of research and provided an implementation of his parsing algorithm in code that is easy to understand and to extend. Yishai Feldman contributed ideas in many technical discussions, particularly, on the analysis of loops.

I've gained many useful technical insights and writing suggestions from Chuck Rich and Richard Waters. Other readers who have provided helpful feedback are Dave Chapman, Yishai Feldman, Howard Reubenstein, and Scott Wills.

I am grateful for my friends, especially Jean Moroney, Elizabeth Turrisi, Anuja Kohli, and Anita Killian, who gave me moral support and many happy times. I am lucky to have wonderful brothers and sisters, who make going home fun and being at MIT bearable.

I'd like to thank Scott, my husband, for making sure I finished my thesis and for making me very happy always.

I am thankful for the love and support of my parents to whom this thesis is dedicated.

Contents

Chapter 1

Introduction

Typically, programmers do not attempt to understand a program by collecting facts and proving theorems about it, except as a last resort. Much of the programmer's understanding comes from recognizing familiar parts and hierarchically building an understanding of the whole based on the parts. For example, a programmer may recognize that a bubble sort algorithm is being used to order the elements of a table. The table may be recognized as having been implemented as a linked list of entries. This model of program understanding, called *analysis by inspection*, has been developed by Rich [28].

The process of identifying occurrences of familiar algorithmic fragments and data structures in a program is referred to in this report as *program recognition.* A system, called the Recognizer, is presented which performs program recognition automatically. Given a program, the Recognizer builds a hierarchical description of the stereotypic computational fragments and data structures of which the program is constructed.

Such a description of a program may be useful in activities such as documenting, maintaining, modifying, and debugging the program. Automatic program recognition can help make these tasks easier to carry out, both manually and automatically.

Aside from its practical applications, program recognition is a worthwhile problem to study from a theoretical standpoint in Artificial Intelligence. It is a step towards modeling how human programmers understand programs based on their accumulated programming experience. It is also a problem in which the representation of knowledge is the key to the efficiency and simplicity of the techniques used to solve the problem.

The Recognizer's Approach to Program Recognition

Performing program recognition automatically is difficult. Typically, there are a variety of ways to syntactically express computational structures in code. Some structures are diffuse in nature, requiring a global view of the code in order to find them.

The Recognizer's approach to program recognition differs from current approaches in two important ways. First, it does not deal with the program in its source code form. Rather, it uses a representation for programs which evolved from the Plan Calculus of Rich, Shrobe, and Waters ([27,28,31,41]). The Plan Calculus is a programming language-independent representation in which programs are represented as graphs, called *plans*. In a plan, nodes stand for operations, and edges specify the data and control flow between them. Plans are quasi-canonical in the sense that they abstract away from how data and control flow are implemented. It doesn't matter which binding or control constructs are used in a program – only net data and control flow is shown in the arcs. This allows stereotypic structures to be recognized in programs that have much syntactic variability without having to anticipate all possible syntactic variations of each stereotypic structure and without using transformations to canonicalize the code.

The second difference between the Recognizer's approach and other approaches is that instead of using heuristics, the Recognizer employs an algorithmic technique in which program recognition is treated as a parsing task. The basic idea is to convert the program into the graph representation (which evolved from the Plan Calculus), translate the library of familiar structures to be recognized into a graph grammar, and then to parse the program in accordance with the grammar. Because the entire graph is systematically parsed, this approach is more exhaustive than approaches which use heuristics. Also, because the Recognizer analyzes the program in terms of a grammar of structures, when the structures are found, a hierarchical description of how they are related to each other can be given. The heuristic methods cannot easily generate this higher level description.

The Recognizer's approach is based on the idea that the stepwise refinement of a program can be modeled as a formal grammar derivation. Brotsky [3] drew an analogy between the implementation of high-level programming operations in terms of lower-level operations and the derivation of a graph by systematically replacing given nodes with specified subgraphs. If programs are represented as graphs and allowable implementation steps are captured in grammar rules, then a parse of the program's graph provides a description of the program's top-down design.

No claim is being made here that programmers actually use top-down derivation in designing a program. Nor is it being claimed that formal parsing is a psychologically valid model of how programmers understand existing programs. For the present work, a grammar is simply a useful way to encode the programmer's knowledge about programming so that parsing can be used for program recognition. A problem for future research is to develop more psychologically valid models. The parsing model is a good place to start.

Overview of the Recognizer

As a first step towards applying the parsing metaphor to program recognition, Brotsky [4] developed a general-purpose graph parser which operates on a restricted class of labeled, acyclic, directed graphs, called *flow graphs.* His flow graph parser takes as input a flow graph and a flow graph grammar and determines whether or not the graph can be generated from the grammar. If it can, it gives all the possible derivations. The current research is concerned with the application of the flow graph parser to program recognition. In order to use this parser, a program must be treated as a flow graph and the collection of stereotypic structures must be treated as a flow graph grammar.

In converting the program to a flow graph, the Recognizer first translates the program source code to a plan by performing a control and data flow analysis on the program (as is shown in Figure 1.1).

Although the program is a graph when represented as a plan, it is not quite a *flow* graph. There are several differences between the two representations. For example, plans may contain cycles while flow graphs are acyclic. Data flow arcs may fan out to several nodes within plans, but flow graphs cannot contain fan-out of arcs.

To make up for the differences between the two representations, some features, such as fan-out arcs, are dealt with by extending the flow graph formalism and parser. Features of the Plan Calculus which cannot be handled by extending the parser, such as cycles, are transferred to *attributes* on the nodes and edges of the flow graph. These are dealt with by a reasoning mechanism which is separate from the parser.

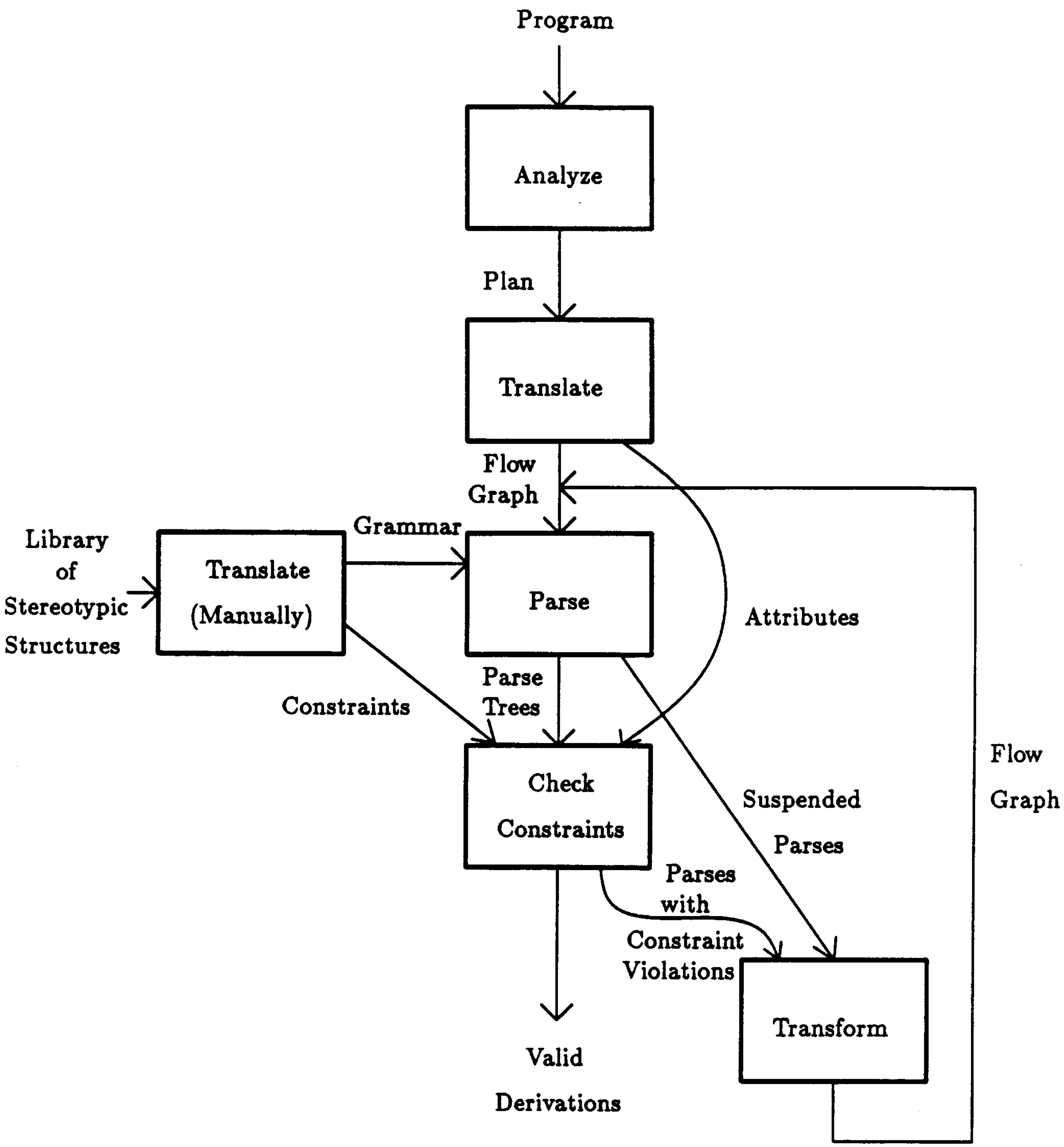


Figure 1.1: The Logical Structure of the Recognizer

In order to convert information into attributes, an intermediate translation stage occurs between the flow analysis and parsing stages. (See Figure 1.1.) The flow graph which results from this translation contains a subset of the plan's nodes and edges. Any information that the plan contains but which is not in the flow graph's structure is stored in the attributes of the flow graph.

Not only must the program be represented as a flow graph in order to use the parser for program recognition, but also the library of stereotypic structures must be translated into a flow graph grammar. The grammar rules created by this manual translation place *constraints* on the attributes of the program's flow graph. The constraint checking phase of the Recognizer filters out the parses which do not satisfy the constraints. The translation of the library into grammar rules therefore involves similar issues of moving information from plans into constraints.

Figure 1.1 shows that the translation of the library to a flow graph grammar is done manually. However, there is no intrinsic reason why the translation must be done by hand. The same analyzer used in translating the program to a plan could be used if suitably extended. Because of time constraints, the library is currently translated manually into a grammar. This isn't a problem because this translation must be done only once for the library.

There are two major obstacles to performing program recognition via parsing. The first difficulty is that recognizing some familiar structures requires that the program's plan be seen in more than one way. (An example of this will be seen when the recognition of plans using constants is discussed.) A mechanism is used which transforms subgraphs of the program's graph into alternative subgraphs. The modified graph can then be reparsed. The transformation is done in the Transform phase shown in Figure 1.1. This will be explained in more detail later.

The second reason program recognition cannot be performed simply by parsing the program's graph is that programs are rarely constructed entirely out of familiar structures. There are bound to be unrecognizable sections in most large and complex programs. Recognition will therefore be *partial*. This means that parses may not always complete, i.e., they won't always reduce the program's graph to a start node. However, the Recognizer gathers useful information from successful subparses. Partial recognition also requires that the Recognizer ignore parts of the program. The Recognizer does this by starting parsers at all "points" in the program's graph. (The technical notion of this will be explained in section 2.3.4.) The parsers are allowed to run as far as they can in order to extract as much information as possible.

In summary, the research described in this report focuses on the following areas:

- Developing a flow graph projection of the Plan Calculus.
- Adapting the flow graph parser to parse an extended flow graph representation.
- Coordinating parsers started at all points throughout the program graph.
- Dealing with information which is converted to attributes in the translation from plan to flow graph.
- Allowing the program's plan to be seen from multiple points of view in order to recognize more features.

Motivation

This research is part of the Programmer's Apprentice (PA) project ([31,32,33,44]) whose goal is to design an intelligent software development system for assisting expert programmers in all aspects of programming. Many of the tools brought together in the PA will benefit from the use of a program recognition module.

The Recognizer will also be very useful in the area of computer-aided instruction (CAI), since its ability to abstract away from the syntactic features of a program allows it to deal with the great variability inherent in student programs. The ways that the Recognizer can assist CAI applications as well as programming tools will be discussed further in Chapter 3.

Organization

Chapter 2 describes the problem of program recognition, giving the goals of the recognition system and describing the parsing technique used to achieve them. This includes discussing the rationale behind using parsing, how flow graph parsing works, and what additional mechanisms are needed to apply parsing to program recognition. Chapter 3 discusses future work anticipated in improving the Recognizer. The chapter also shows how automatic program recognition can be put to use in a variety of applications. Chapter 4 discusses related approaches to automating the recognition process.

Chapter 2

Program Recognition via Parsing

Typically, in trying to understand code, a programmer looks for familiar algorithmic fragments or data structures in it. Following Rich, Shrobe, and Waters ([28,31,32,41]), these familiar structures will be referred to as *clichés* . Suppose, for example, that an experienced programmer were given the code shown in Figure 2.1 and asked to describe it. (The code is written in Common Lisp, as is all of the code in this report.) Though contrived, this example is useful because it displays several clichés in a short example. The programmer might recognize that HT-MEMBER is checking for membership of ELEMENT in STRUCTURE which is a hash table. Since one of the ways to implement a set is as a hash table, the programmer might see HT-MEMBER as a set membership operation. The programmer knows that STRUCTURE is a hash table because a bucket is fetched by computing a hash code and using it as an index into an array. The bucket is then searched for ELEMENT. If found, T is returned, otherwise NIL is the result. Because the search for ELEMENT in the bucket is terminated as soon as an element is found which is lexicographically greater than ELEMENT, the programmer might assume that each bucket is implemented as an ordered list. This also indicates that the data objects in the hash table are strings and are arranged in lexicographic order within each bucket.

2.1 Overview of Recognizer's Capabilities

The Recognizer is able to automatically perform a recognition process similar to the one just described. The Recognizer takes as input a program and a library of clichés and finds all

```
(DEFUN HT-MEMBER (STRUCTURE ELEMENT)
  (LET ((BUCKET (AREF STRUCTURE (HASH ELEMENT)))
        (ENTRY NIL))
    (LOOP DO
          (IF (NULL BUCKET) (RETURN NIL))
          (SETQ ENTRY (CAR BUCKET))
          (COND ((STRING> ENTRY ELEMENT) (RETURN NIL))
                ((EQUAL ENTRY ELEMENT) (RETURN T)))
          (SETQ BUCKET (CDR BUCKET)))))
```

Figure 2.1: Undocumented Common Lisp Code

instances of the clichés in the program.

As a way of demonstrating the effectiveness of the analysis, one output the Recognizer can automatically produce is a kind of program documentation. This documentation is generated by a module which receives all valid derivations from the Recognizer, as is shown in Figure 2.2. The documentation module uses a technique, due to Cyphers [8], of associating with each cliché a schematized textual explanation fragment. When a cliché is recognized, the slots of the explanation fragment are filled in by names of functions, variables, or other clichés found. Given the code in Figure 2.1, the Recognizer will produce the following comments:

```
HT-MEMBER is a Set Membership operation.
  It determines whether or not ELEMENT is an element of
  the set STRUCTURE.
The Set is implemented as a Hash Table.
  The Hash Table is implemented as an Array of buckets,
  indexed by hash code.
    The buckets are implemented as Ordered Lists.  They
    are ordered lexicographically.  The elements in the
    buckets are strings.  An Ordered List Membership is
    used to determine whether or not ELEMENT is in the
    fetched bucket, BUCKET.
```

Since the Recognizer does not understand natural language, some of the text generated may be awkward or choppy. Fixing awkward text is not part of this research. Even though the documentation produced by the Recognizer may not be as smooth as that written by

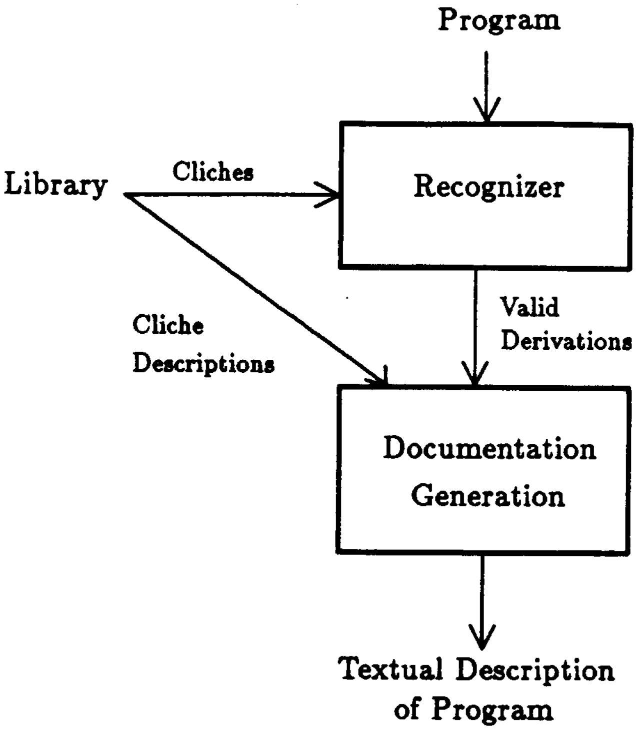


Figure 2.2: Generating Textual Descriptions from the Recognizer's Output

a programmer, it describes the main features of the program's design. A desideratum for judging this output is for an experienced programmer to be able to understand it easily.

2.1.1 Recognizing Clichés in Wide Classes of Equivalent Programs

An essential capability of the Recognizer is that it can recognize when two programs are semantically equivalent, i.e., they describe the same computation (see [27]), even though they might differ either syntactically or in the way they implement a high-level operation.

Figure 2.3 shows another version of the HT-MEMBER code which is significantly different syntactically from the code in Figure 2.1. The Recognizer yields the same analysis and textual description of HT-MEMBER2 as was given for HT-MEMBER. The two programs differ in the control and binding constructs which are used and in the binding of intermediate results. The language-independent, quasi-canonical form into which the programs and clichés are translated allows the recognition process to yield the same description of both programs without being misled by the syntactic differences.

Two programs may also be seen as doing an equivalent operation even though they implement it in entirely different ways. The Recognizer will give the same high-level description of

```
(DEFUN HT-MEMBER2 (STRUC ELEM &AUX BKT)
  (SETQ BKT (AREF STRUC (HASH ELEM)))
  (LOOP DO
    (WHEN (NULL BKT) (RETURN NIL))
    (LET ((ENTRY (CAR BKT)))
      (IF (STRING> ENTRY ELEM)
          (RETURN NIL)
          (IF (EQUAL ENTRY ELEM)
              (RETURN T)
              (SETQ BKT (CDR BKT)))))))
```

Figure 2.3: A Syntactic Variation of HT-MEMBER

```
LIST-MEMBER is a Set Membership operation.
The set is implemented as a List.
  A List Membership is used to determine whether
  ELEMENT is an element of the set STRUCTURE.
(DEFUN LIST-MEMBER (STRUCTURE ELEMENT)
  (LOOP DO
    (COND ((NULL STRUCTURE) (RETURN NIL))
          ((EQ (CAR STRUCTURE) ELEMENT)
           (RETURN T))
          (T (SETQ STRUCTURE (CDR STRUCTURE))))))
```

Figure 2.4: A Different Implementation of a Set Membership Operation

both of the programs. The rest of the description for each program will be different, reflecting the differences in implementations on lower levels. For example, the code in Figure 2.4 can be seen as equivalent to that of Figures 2.1 and 2.3 when viewed on a higher level of abstraction. They all are described as set membership operations, even though Figures 2.1 and 2.3 are *Hash Table Membership* tests and Figure 2.4 is a *List Membership Test.*

2.1.2 Overlapping Implementations

Another capability of the Recognizer is that it is able to recognize two distinct operations in code, even though their implementations may overlap. This is an important strength since it allows optimized code to be analyzed. For example, consider the following program.[1]

```
(DEFUN MAX-MIN (L)
  (LET ((MAX (CAR L))
        (MIN (CAR L))
        (L (CDR L)))
    (LOOP DO
      (COND ((NULL L) (RETURN (CONS MAX MIN))))
      (SETQ ELEM (CAR L))
      (IF (> ELEM MAX) (SETQ MAX ELEM))
      (IF (< ELEM MIN) (SETQ MIN ELEM))
      (SETQ L (CDR L)))))
```

This program computes both the maximum and the minimum of a non-empty list of numbers. Separately these are both standard computations. They are similar in that they both enumerate the elements of the input list. The programmer has exploited this commonality by sharing the list enumeration between the two higher-level operations. This type of optimization is common and essential to good programming. It is therefore a valuable feature of the Recognizer that it is able to analyze the MAX-MIN program even though the resulting decomposition will not be strictly hierarchical. In the case of MAX-MIN, the decomposition may be represented as in Figure 2.5. The diagram's top node corresponds to the program itself. The second level indicates that the program may be seen as consisting of a *Maximum* operation and a *Minimum* operation. Each of these operations are further decomposed into an initialization (which is (CAR L)), an *Accumulation* which keeps track of the largest (or

[1] Taken from an example in a similar discussion in Chapter 1 of [28].

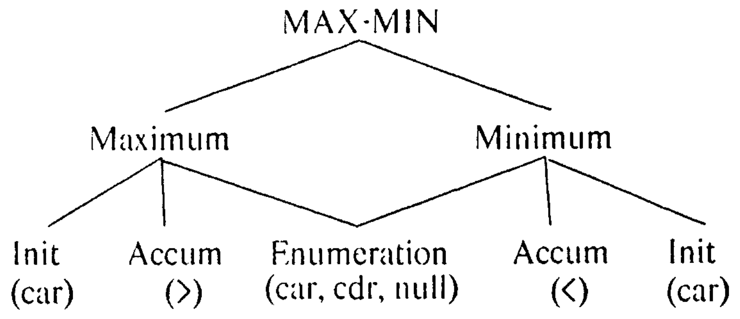


Figure 2.5: Parse Tree for the MAX-MIN program

smallest) element found so far, and an *Enumeration* which is shared by the two subtrees for Maximum and Minimum. The plans in this decomposition will be discussed further later.

2.1.3 Limits on What the Recognizer Understands

The Recognizer is able to automatically analyze programs containing nested expressions, conditionals, single- and multiple-exit loops, and some data structures. The Recognizer cannot handle side effects other than assignment to variables, nor can it analyze programs containing recursion, arbitrary data abstraction, or functional arguments. The library of clichés given to the Recognizer is a subset of an initial library of clichés which has been collected and formalized by Rich [28]. The entire library cannot be used because some clichés contain characteristics not yet handled by the Recognizer, such as side effects and recursion.

Another limitation of the Recognizer is that it cannot explain what a piece of code is good for. For example, experienced programmers usually know what kinds of operations are made more efficient by implementing an abstract data structure one way as opposed to another. For instance, a hash table implementation of a set is expected to be more efficient than a list implementation if frequent membership testing or searching is to be performed on it. The Recognizer is not able to make this observation since that involves a much deeper understanding of the clichés than is captured in the grammar. Clichés may be annotated with canned observations about a particular implementation if necessary, but the Recognizer doesn't understand that type of comment.

The Recognizer also does not extract information from variable names or any documentation surrounding the original program (e.g., it can't guess that the abbreviation "HT" stands for "hash-table").

2.2 The Parsing Metaphor

A good way to explain the design of a program is to give a top-down account. Higher-level operations may be described as being implemented in terms of lower-level operations on each successive level of abstraction. This is analogous to a flow graph derivation process wherein a set of rewriting rules, called a flow graph grammar, is used to specify how given nodes can be replaced by specified subgraphs. In this analogy, flow graphs correspond to graphical abstractions of programs, flow grammars specify allowable implementation steps, and the resulting parse tree gives the top-down design. The Recognizer's approach is based on this analogy.

A crucial constraint on the flow grammar in this analogy is that it be *context-free*. The left-hand side of each rule is a single node which gives a more abstract description of the right-hand side graph. If the rules were context-sensitive, the left-hand side would typically be another graph instead of a single node. This rule specifies a series of substitutions or modifications to be made to the graph being derived. Context-sensitive rules would, by analogy, specify program transformations in a wide-spectrum language, rather than incremental refinement steps.

As will be seen later, the Recognizer makes use of program transformations in allowing a program to be seen from multiple points of view (e.g., in the recognition of plans using constants). However, these rules are not used in the normal parsing process. They are applied to the program's plan in the transformation phase and then parsing is restarted on the modified graph.

The key to being able to raise parsing from being simply a metaphor to being a technique for recovering a top-down design of a program is the Recognizer's representation of programs. This point will become clearer once the Plan Calculus, the cliché library, and the flow graph representation have been described in more detail.

2.2.1 The Plan Calculus

The representation of knowledge has a direct effect on how easy or hard it is to carry out a complex operation on that knowledge. In trying to perform the task of automated program recognition, this is certainly the case. The Plan Calculus makes the recognition task much easier and more straightforward. It allows programs to be represented as graphs and the cliché

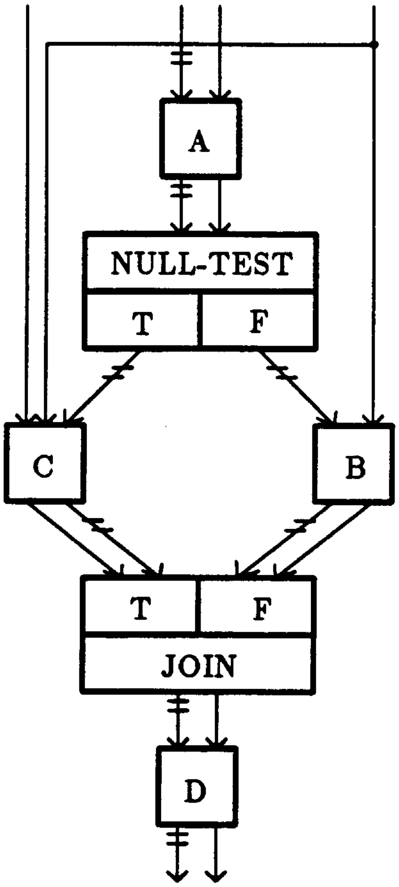


Figure 2.6: The Plan for SOME-FUNCTION

library as a grammar, making them more amenable to a parsing technique for recognition. The Plan Calculus representation is also valuable because it abstracts away from the syntactic features of a program, allowing the program's algorithmic structure to be emphasized. Another important fact about the Plan Calculus is that it is programming language-independent and therefore may be used as the target representation for many different languages (e.g., Lisp [31], Cobol [12], Fortran [41], and Ada [44]).

A program is represented in the Plan Calculus as a graph, called a *plan*, in which nodes represent operations, and edges explicitly show the control and data flow between them. As an example, Figure 2.6 shows the plan representation for the following code.

```
(DEFUN SOME-FUNCTION (X Y Z)
  (D (COND ((A Y) (B Z))
           (T (C X Z)))))
```

As is shown in the figure, there are two types of edges in plans: one indicating data flow and the other control flow. (When a graph is drawn, control flow edges are denoted by cross-hatched lines, while data flow edges are solid lines.) Branches in control flow caused by a conditional are represented by nodes, called *splits*, which have one control flow input arc and two control flow output arcs, one for each of the cases of the split (true or false). The type of all split nodes (in the present work) is NULL-TEST since the only primitive determiner for conditional branching in Lisp is the test to see if an object is NIL. In other languages, there may be more types of primitive tests and therefore more types of splits. A merging of control flow and data flow (e.g., to obtain the return value of a conditional), is represented by a *join*. These are boxes which have two or more incoming control flow arcs.

There is no fan-in of control or data flow in plans. Merges are accomplished via joins. Also, data flow may fan out, but control flow may not.

In the Plan Calculus, all syntactic information about what types of binding or control constructs are used is abstracted away. It doesn't matter if one program uses COND while another uses IF, for example, or if one program binds intermediate results in variables while another passes all data through nested expressions. If all the data flow is coming from matching places in each program and the same operations are performed on the data, the plans for the programs will be equivalent in terms of data flow. This is because the representation shows *net* data flow. For instance the following code will have the same plan SOME-FUNCTION has (shown in Figure 2.6) even though it is syntactically very different from SOME-FUNCTION.

```
(DEFUN ANOTHER-FUNCTION (X Y Z)
  (LET* ((BZ (B Z))
         (CXZ (C X Z))
         (RESULT (IF (A Z) BZ CXZ)))
    (D RESULT)))
```

Net control flow is also represented, so that the particular control flow constructs used by the program are abstracted away, making it easier to compare two programs. However, two equivalent programs might not have exactly the same plan because some of the control flow information of the program is not canonicalized enough by the plan. As will be shown, this is a major source of trouble in trying to apply parsing to recognition.

Loops

Loops may be represented in the Plan Calculus in two different ways, either as a cycle in control and data flow arcs, or by using a recursive node. The Recognizer uses the cyclic representation of loops for reasons which will be explained in section 2.3.3. As an example of the cyclic representation, the plan for the following code which computes the length of a list is given in Figure 2.7.

```
(DEFUN MY-LIST-LENGTH (L)
  (LET ((COUNTER 0))
    (LOOP DO
      (COND ((NULL L) (RETURN COUNTER)))
      (SETQ L (CDR L))
      (SETQ COUNTER (1+ COUNTER)))))
```

The second way that loops may be represented is the same way that a recursive call to a function is represented, i.e., as a node with the name of the recursive function as its operation type. To represent a loop using recursive nodes, the code containing the loop must be translated into its tail-recursive form. For instance, the function MY-LIST-LENGTH would be translated into:

```
(DEFUN RECURSIVE-LIST-LENGTH (L)
  (LABELS ((INTERNAL-LIST-LENGTH (L COUNTER)
             (IF (NULL L)
                 COUNTER
                 (INTERNAL-LIST-LENGTH
                   (CDR L)
                   (1+ COUNTER)))))
    (INTERNAL-LIST-LENGTH L 0)))
```

The plan for the tail-recursive version of MY-LIST-LENGTH is given in Figure 2.8. The node whose label is "RECURSIVE-LIST-LENGTH" is a recursive node.

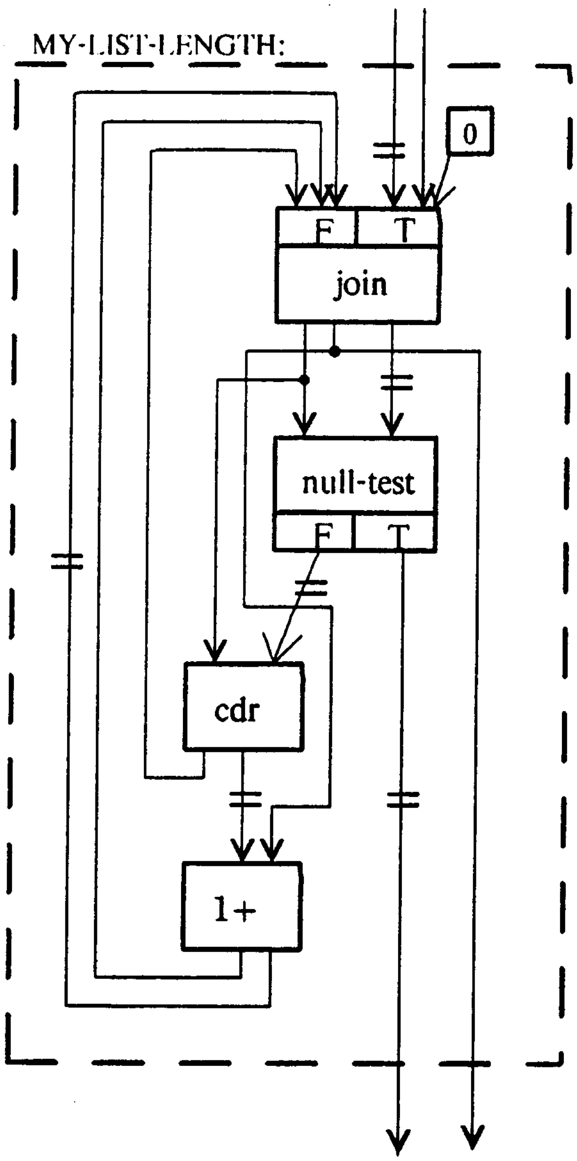


Figure 2.7: The Plan for MY-LIST-LENGTH

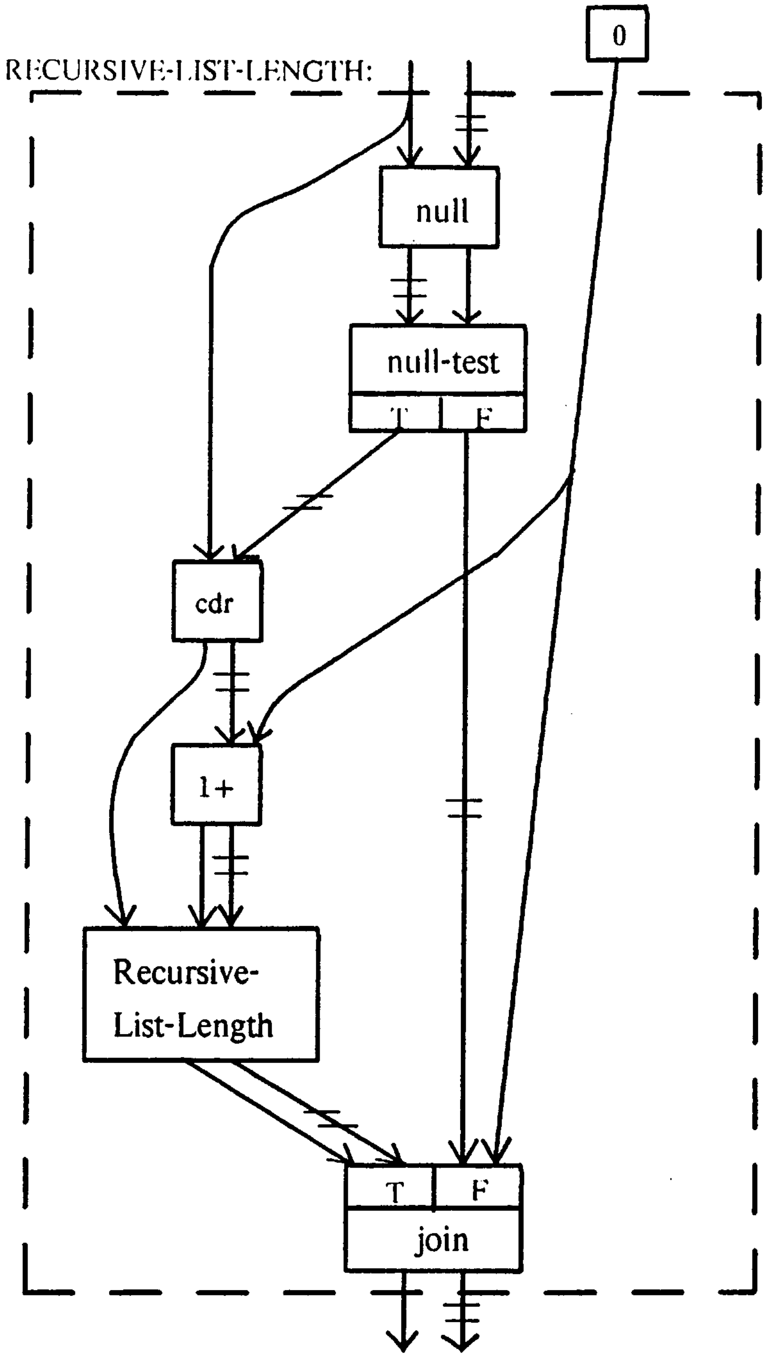


Figure 2.8: The Plan for the Tail Recursive Program RECURSIVE-LIST-LENGTH

The Flow Analyzer

The module within the Recognizer which converts a program from source code to a plan is the Flow Analyzer. The method of translation is borrowed from the analysis techniques employed by KBEmacs [44], the most recent demonstration system implemented as part of the Programmer's Apprentice project. The translation is done in two stages: macro-expansion, followed by control and data flow analysis.

The macro-expander translates the program into a simpler language of primitive forms. It also selectively flattens parts of the program by open-coding functions and procedures inside their callers. This allows variability in the way programs to be analyzed are broken down into subroutines. It does not automatically flatten the entire program for efficiency reasons and also because recursive functions cannot be completely open-coded.

The control and data flow analysis is performed by a symbolic evaluation of the program. The evaluator follows all possible control paths of the program, converting operations to nodes and placing edges corresponding to data and control flow between operations. Whenever a branch in control flow occurs, a split is added. Similarly, when control flow comes back together, a join is placed in the graph and all data representing the same variable is merged together.

The graph that results from the flow analysis is a more canonical, language-independent form of the program than the code is. Since it contains no deeper knowledge about the program, it is called a *surface plan.*

The flow analyzer used by the Recognizer translates Lisp programs into plans. Similar analyzers have been written for subsets of Cobol [12], Fortran [41], and Ada [44]), but are not used in this system.

2.2.2 The Cliché Library

The cliché library contains a taxonomy of standard computational fragments and data structures represented as plans. For example, Figure 2.9 shows a plan which defines the cliché for finding the length of a list.

Besides the vocabulary of standard forms, the plan library contains implementation relationships between the forms. The implementation relationships specify how one standard

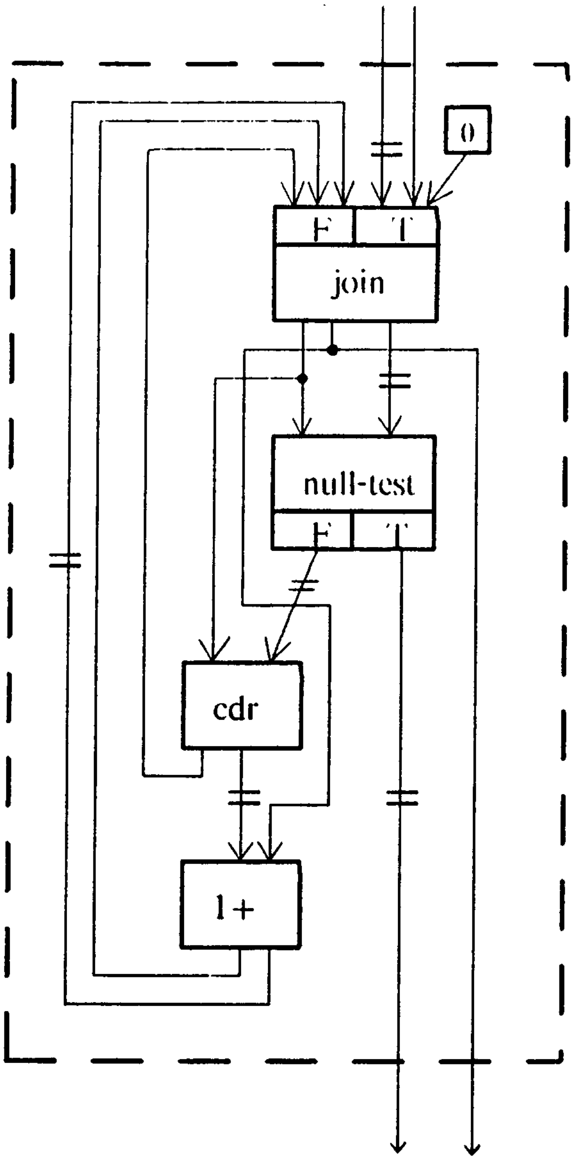


Figure 2.9: A Definition of the Cliché for Computing a List's Length

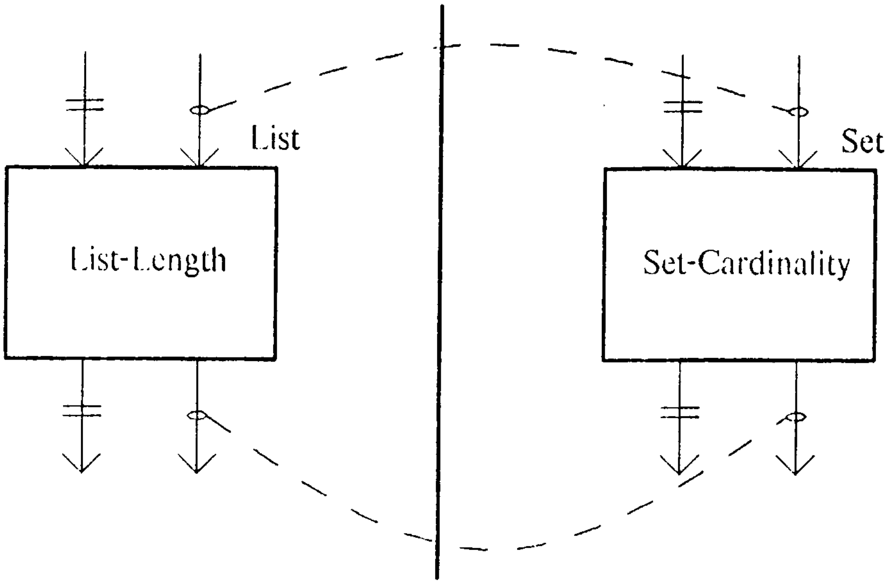


Figure 2.10: Overlay Showing List-Length Acting as Set-Cardinality

form may be implemented in terms of others. They are expressed in the Plan Library by *implementation overlays.* They represent a shift in the way an operation or structure is viewed by a programmer. For example, a typical overlay is **list-length>set-cardinality** (read "list-length as set-cardinality") which specifies that a list length operation can be viewed as an operation to find the size of a set, given that the input set is implemented as a list. The **list-length>set-cardinality** overlay is shown in Figure 2.10.

As will be seen, both plan definitions and implementation overlays are (manually) translated into, or *induce*, grammar rules which are used to parse programs. Plan definitions become rules which take a single node to a graph containing any number of nodes, while an overlay-induced rule takes a single node to a graph containing exactly one node. When a plan definition rule is used in the parse of a program, it recognizes a group of lower-level operations, tests, and data structures as a single higher-level one. When an overlay-induced rule is used, it uncovers a design decision which implements a certain abstract data type as a lower-level data type.

This is only the intuition behind the Recognizer's approach. The actual grammar rules used by the Recognizer do not contain the plan diagrams in the form shown in Figures 2.9 and 2.10. Nor is the program parsed as a plan. This is because plans are not flow graphs as required by the Recognizer's flow graph parser. The differences between the flow graph representation and plans will become apparent when flow graphs are described in the next section.

The reasons the flow graph parser is used by the Recognizer are that it is efficient, it works, and it is extendable. Brotsky's algorithm runs in time polynomial in the number of nodes in the input graph and linear in the size of the grammar. Brotsky provided a working implementation of the algorithm written in Lisp. The parser is also easily extendable, so that most of the differences between plans and flow graphs may be eliminated. Those that are not are dealt with by additional mechanisms which are built on top of the parser. These mechanisms will be discussed after the extensions made to the parser are described.

In addition to extending the definition of flow graphs, some of the information contained in the plan representation of programs is removed from the plan and converted to attributes. Thus, the representation of a program is a flow graph *projection* of the program's plan. It is a flow graph whose structure contains some, but not all, of the information the plan contains.

2.2.3 Flow Graph Parsing

A flow graph is a labeled, acyclic, directed graph with the following restrictions on its nodes and edges. (This definition is due to Brotsky and will be extended.)

1. Each node has a label, called its *type*.

2. Each node has *input ports* and *output ports*. (Ports are positions at which edges enter or leave a node.) Each port is labeled. No port can be both an input and an output port. There are the same number of input and output ports on all nodes of the same type.

3. There is at least one input and one output port on each node.

4. All edges run from a particular output port of one node to a particular input port of another. No port may have more than one edge entering or exiting it. Therefore, a node can be adjoined by at most as many edges as it has ports.

A further characteristic of flow graphs is that ports need not have edges adjoining them. Any input (or output) port in a flow graph that does not have an edge running into (or out of) it is called an *input* (or *output*) of that graph.

Extending the Flow Graph Formalism

Some of the differences between the Plan Calculus and flow graph representations may be eliminated by extending the flow graph formalism and parser. Some characteristics which occur in plans but which may not occur in flow graphs (given the definition in the previous section) are:

- **fan-out edges** — The results of an operation may be shared by two or more operations. This is represented as edges fanning out of a single port on a node.
- **straight-through arcs** — Data may enter a program and be given as an output without being used by any operations in the program. This is represented by an edge running through a graph which does not come from any output port or into any input port.

In addition, some features do not occur in plans, but arise in the flow graph projection of plans as a result of solutions to other problems (as will be explained in Section 2.3.1). These are:

- **fan-in edges** — More than one edge may enter the same input port on a node.
- **sinks** — Nodes may have no output ports.

The parser has been extended so that flow graphs may contain any of these characteristics. In particular, the third and fourth restrictions given in the definition of a flow graph have been lifted. Removing the third restriction allows flow graphs to contain sinks. Lifting the fourth restriction means more than one edge may fan into or out of a port. It also means straight-through edges are no longer forbidden.

An example of a flow graph is given in Figure 2.11. The ports and edges have been labeled with subscripted "p"s and "e"s, respectively, so that they may be referred to. Edge e_1 is a straight-through edge. Edges e_3 and e_4 fan out of port p_2 on node **b**. Edges e_5 and e_8 fan into port p_1 on node **g**. Node **d** is a sink.

A feature of plans but not of flow graphs (even in the extended formalism) is that they may contain cycles. This characteristic is dealt with by higher-level mechanisms built on top of the parser. This will be discussed in a later section (see section 2.3.3).

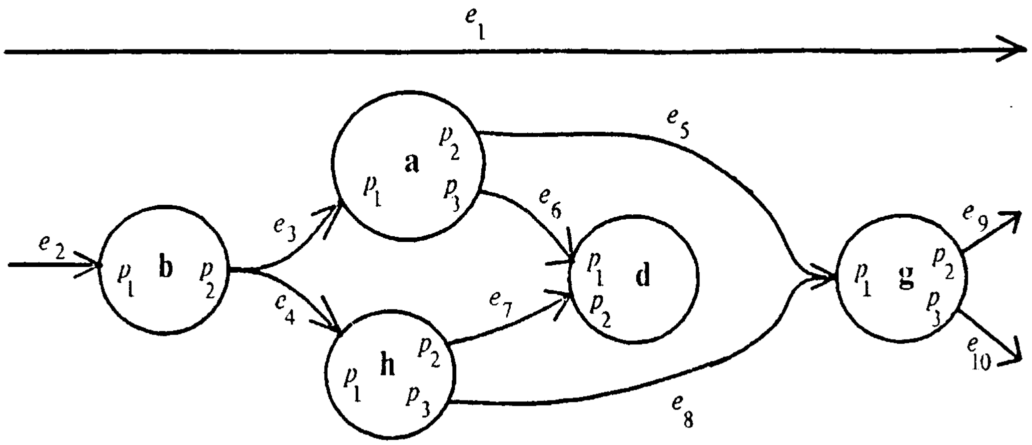


Figure 2.11: A Typical Extended Flow Graph

Flow Graph Grammars

A flow graph grammar is a set of rewriting rules, each specifying how a node in a graph can be replaced by a particular subgraph. All rules in a flow grammar map a single left-hand side node to a right-hand side graph. The left-hand side node is of a nonterminal node-type, while the right-hand side graph can contain nodes of both terminal and nonterminal types. (Throughout this report, nonterminals are denoted by capital letters, while terminals are in lower case.)

The flow graph in Figure 2.11 can be derived from the flow graph grammar shown in Figure 2.12 where node type **A** is a start node type. A sample leftmost derivation of the graph in Figure 2.11 in accordance with the grammar of Figure 2.12 is given in Figure 2.13.

Each rule in a grammar has a mapping between the ports of the left-hand side and the input and output edges of the right-hand side. When shown pictorially, this mapping is indicated by numbers on the ports of the left-hand side node corresponding to edges of the right-hand side graph. The mapping determines the connectivity of the left-hand side when a subgraph matching the right-hand side is reduced to the left-hand side during parsing. In Brotsky's parser, this correspondence is one-to-one. However, the correspondence in the extended parser is both one-to-many and many-to-one.

For example, the rule for **F** in Figure 2.14 contains a one-to-many mapping. The port labeled 3 on **F** is mapped to the output edges of both **f** and **e** on the right-hand side. When

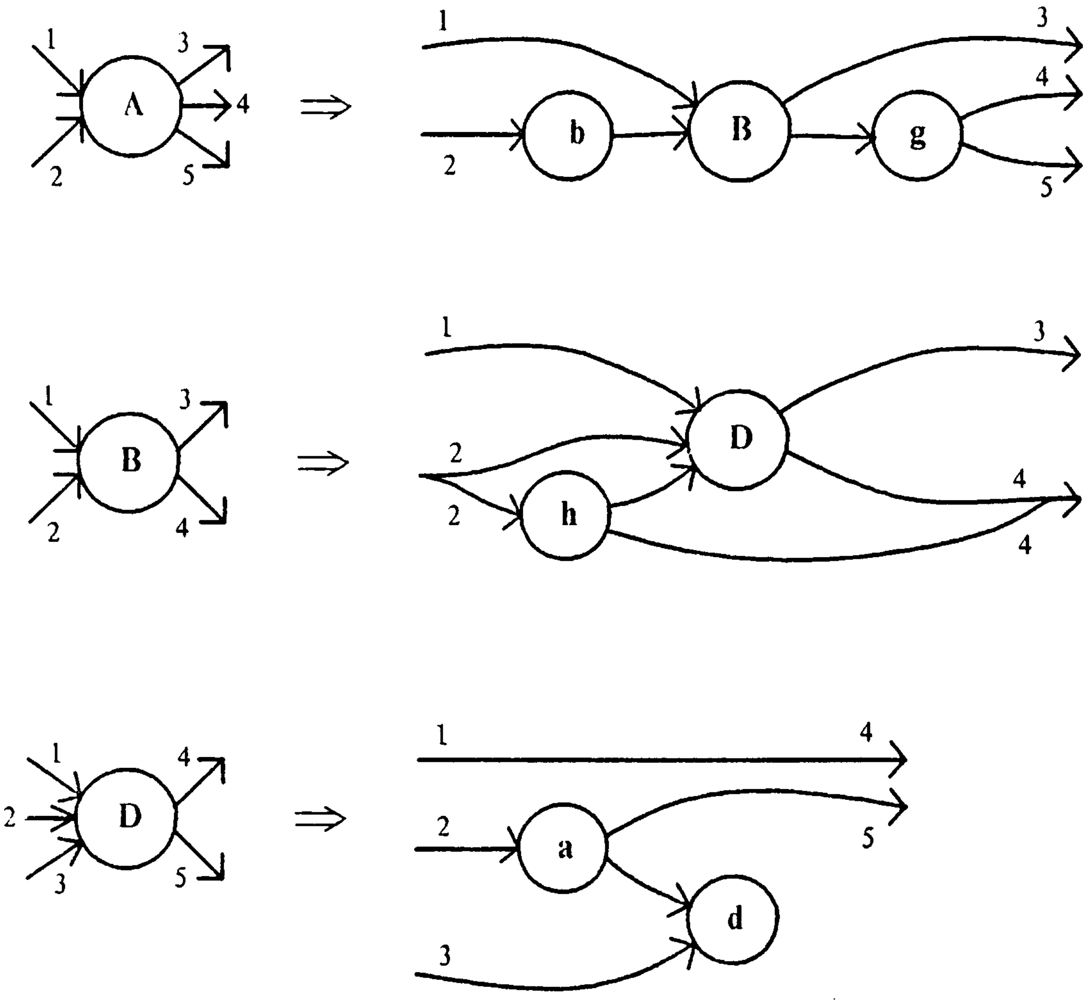


Figure 2.12: A Flow Graph Grammar

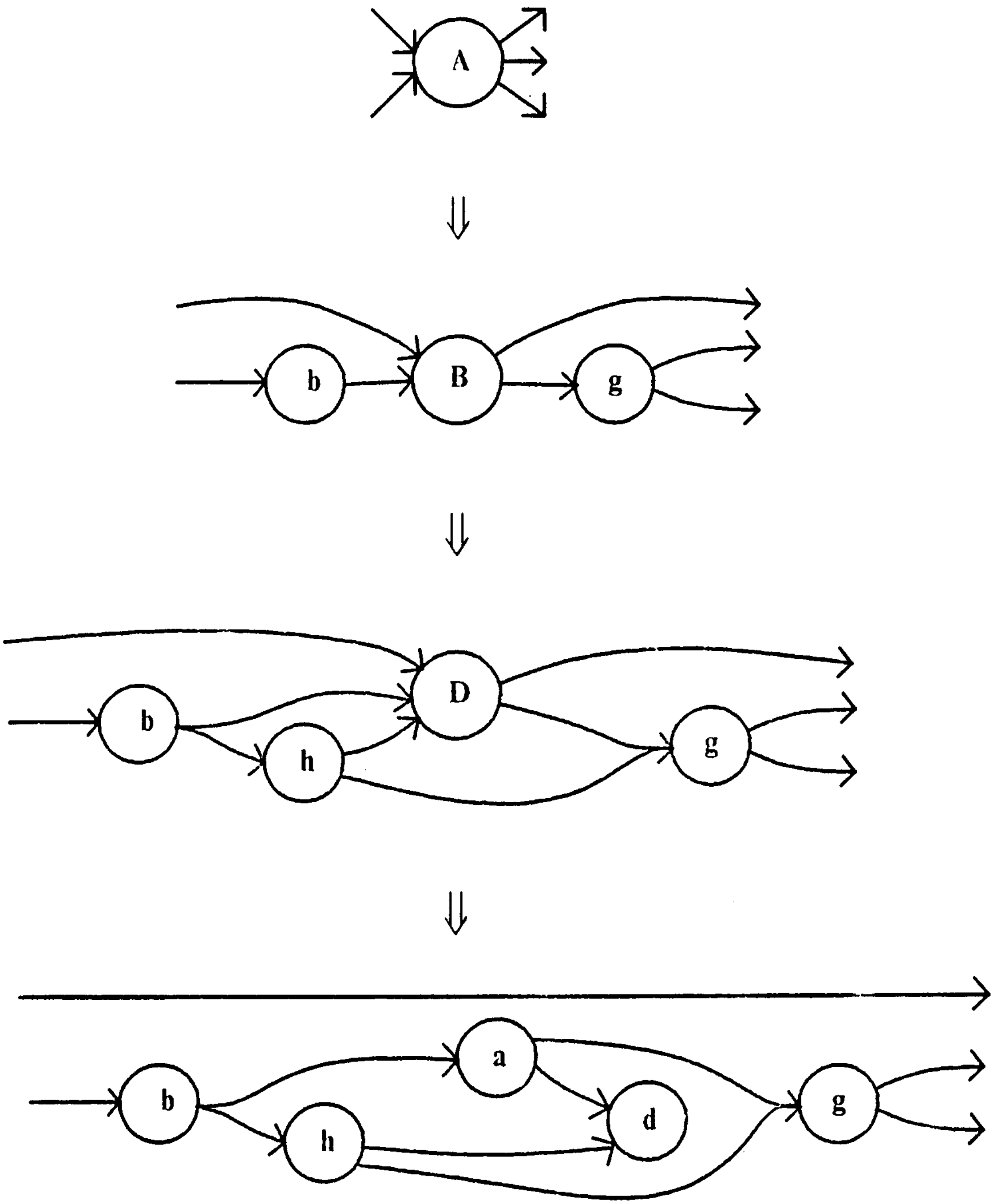


Figure 2.13: A Derivation of a Flow Graph

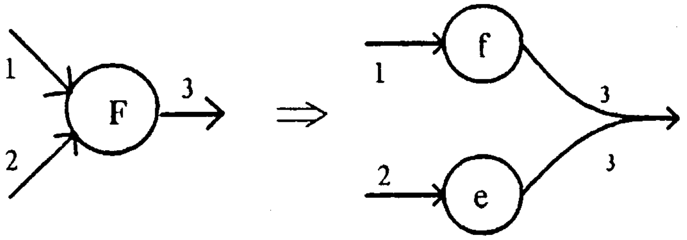


Figure 2.14: A Rule Requiring that a Subgraph's Trailing Edges Fan in

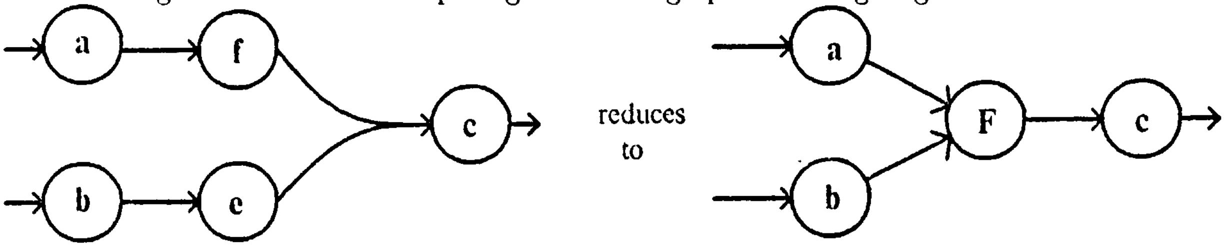


Figure 2.15: Reduction of a Subgraph to **F**

an input (or output) port on the left-hand side node is mapped to more than one edge in the right-hand side graph, the correspondence specifies which input (or output) edges of the subgraph matching the right-hand side must fan out of (or fan into) the same port when the subgraph is reduced to the left-hand side node. For example, if the right-hand side of the rule for **F** were recognized in a graph and reduced to **F**, the input port labeled 1 on **F** will be connected to whatever port **f**'s input edge is connected to. Similarly, **F**'s port 2 connects to wherever **e**'s input edge comes from. Input port 3 will be connected to whatever port both output edges of **f** and **e** are connected to. This is shown in Figure 2.15. If the output edges of **f** and **e** do not fan into some port, the reduction cannot take place.

While a one-to-many mapping requires that certain edges must fan-in or fan-out, a many-to-one mapping specifies a straight-through edge on the rule's right-hand side. The rule for **D** in Figure 2.12 gives an example of a many-to-one mapping. In this rule, the input port labeled 1 and the output port labeled 4 on the left-hand side node **D** are both mapped to the same edge on the right-hand side.

Parsing

Flow grammars derive flow graphs in much the same way as string grammars derive strings. The flow graph parsing process is a generalization of string parsing, in which most of the familiar characteristics of context-free grammars for deriving strings apply. Accordingly, Brotsky developed a flow graph parsing algorithm which generalizes Earley's string parsing algorithm [10]. The following is a much simplified description of the algorithm. The actual operation of the parser is much more sophisticated and optimized. (Consult [4] for more information.)

Brotsky's algorithm deterministically simulates the behavior of a non-deterministic stack-based parser. The simulation essentially allows several graph parsers to run in parallel, each eventually coming up with a different guess as to the derivation of the input. Thus, all possible parses of the input are obtained. The way this is done is by marching a *read head* over the input graph in one pass, scanning all nodes and edges exactly once. The position of the read head is given by the set of edges which it cuts across. For each node scanned by the read head in the input, the algorithm generates all reachable configurations of all parsers being run on the input. Since these parsers are collectively simulating a single nondeterministic parser, the set of all possible configurations of the deterministic parsers can be seen as all the configurations the nondeterministic parser might be in, given the nodes that were scanned in the input graph.

The reachable configurations at each step are recorded in lists of *items*. Each item contains a recognizer which is attempting to match the right-hand side of some grammar rule within the input graph being scanned. The right-hand side to be recognized is called the *target graph* of the recognizer. The item has information, called the *state* of the recognizer, which specifies where in the target graph the recognizer's read head is in relation to the read head in the input graph. When a node is scanned in the input graph, the read head steps over it. The corresponding read head is also stepped in all active items.

Any nonterminal in the right-hand side of a rule causes separate sub-items to be activated for each rule which derives that nonterminal. The item keeps a list of pending calls to these sub-items as well as a list of items to return to when its own recognizer completes.

How Extensions Were Made to the Parser

Because the algorithm is agenda-based, i.e., it works by consulting and updating the current item list, its behavior can be controlled simply by altering this list. This is one of the strengths of the parser which allows it to be adapted and applied to the problem of program recognition. For example, to allow edges to fan out in the input graph or in the grammar rules' right-hand sides, alterations are made to the item-lists as the parser is running. The alterations involve replicating items on the lists so that all possible correspondences between edges in the rule and edges in the graph being parsed are tried.

2.2.4 Additional Mechanisms

Besides making extensions to the flow graph formalism, additional mechanisms have been layered on top of the parser. The mechanisms are useful not only in solving many of the problems of applying parsing to program recognition, but also may be valuable in areas of research besides program understanding.

Attributes

One of the mechanisms built on top of the parser allows attributes to be placed on the nodes and edges of flow graphs to be parsed. This is useful in dealing with information contained in plans which cannot be dealt with easily by the parser. Associated with the right-hand side of each grammar rule is a set of constraints. These are predicates which apply to the attributes of any subgraph structurally matching the rule's right-hand side. During reduction, the constraints are checked (by being evaluated) to be sure that the attributes in the nodes and edges involved obey them. A parse is thrown away if the constraints aren't satisfied. Because the constraint checking is folded into the parsing process, i.e., constraints are checked each time a nonterminal is reduced, invalid parses may be cut off early.

There is a tradeoff between how much information should go into attributes versus into the graph. If too little information is in the graph, a tremendous number of parses may be successful, since the structural information is less specific and restrictive. The burden would then be on the constraint checker to weed out meaningless parses. On the other hand, if too much information is in the graph, the parsing process may become too complicated. For example, it might be harder to canonicalize the graph if too much information were structurally represented in it.

Attribute-Transfer

During reduction, attributes may be transferred from the subgraph which matches a rule's right-hand side to an instance of the rule's left-hand side node. This allows attributes to be propagated up through the grammar. Thus, constraints may be placed on any node on the right-hand side of a rule, including non-terminal nodes.

Each grammar rule has *attribute-transfer specifications* which are used to compute attribute values for the rule's left-hand side node based on the subgraph matching the rule's right-hand side.

As an example, consider the grammar and graph to be parsed in Figure 2.16. The rule for **A** has attribute-transfer specifications which may be used to compute the *color* attribute of **A**. When the rule for **B** is subsequently applied, the constraint on **A**'s color may be evaluated.

The attribute-transfer facility is useful in program recognition. A cliché in the library has not only a definition, but also a set of properties which are true of the more abstract operation it defines. When the cliché is recognized in a program, these properties may be assumed to hold true in the higher-level plans which use the plan recognized. The higher-level plans may, in turn, have constraints on these properties. The properties become attribute values when the plan is translated into a grammar rule. The attribute-transfer mechanism allows the information derived from reduced plans to be propagated to portions of higher-level clichés not yet parsed.

Generalized Node Types

One of the ways in which attributes are used is in allowing generalized node types. When Brotsky's parser compares a node in a grammar rule with a node in the graph being parsed, it checks for the equality of their types. Very often in using parsing to perform program recognition, a grammar rule contains a node which may be one of several different types which together form some identifiable class, for example, the class of all arithmetic operations or the class of all predicates. Rather than providing a separate grammar rule for each type, it is convenient to simply require that a terminal node's type be a member of a particular class. The advantages of this is that the grammar does not become cluttered and fewer parses are generated.

The parser's check for node type equality between a grammar rule's node and a node in the graph being parsed is no longer simply an equality comparison. It is now a constraint, called a ***node-type constraint***, which each node in a grammar rule places on the type of the node being matched with it. An example of a node-type constraint is that a node's type be a member of the set of all arithmetic operators in Common Lisp.

The node-type constraints in the grammar rules are unlike other constraints in that they

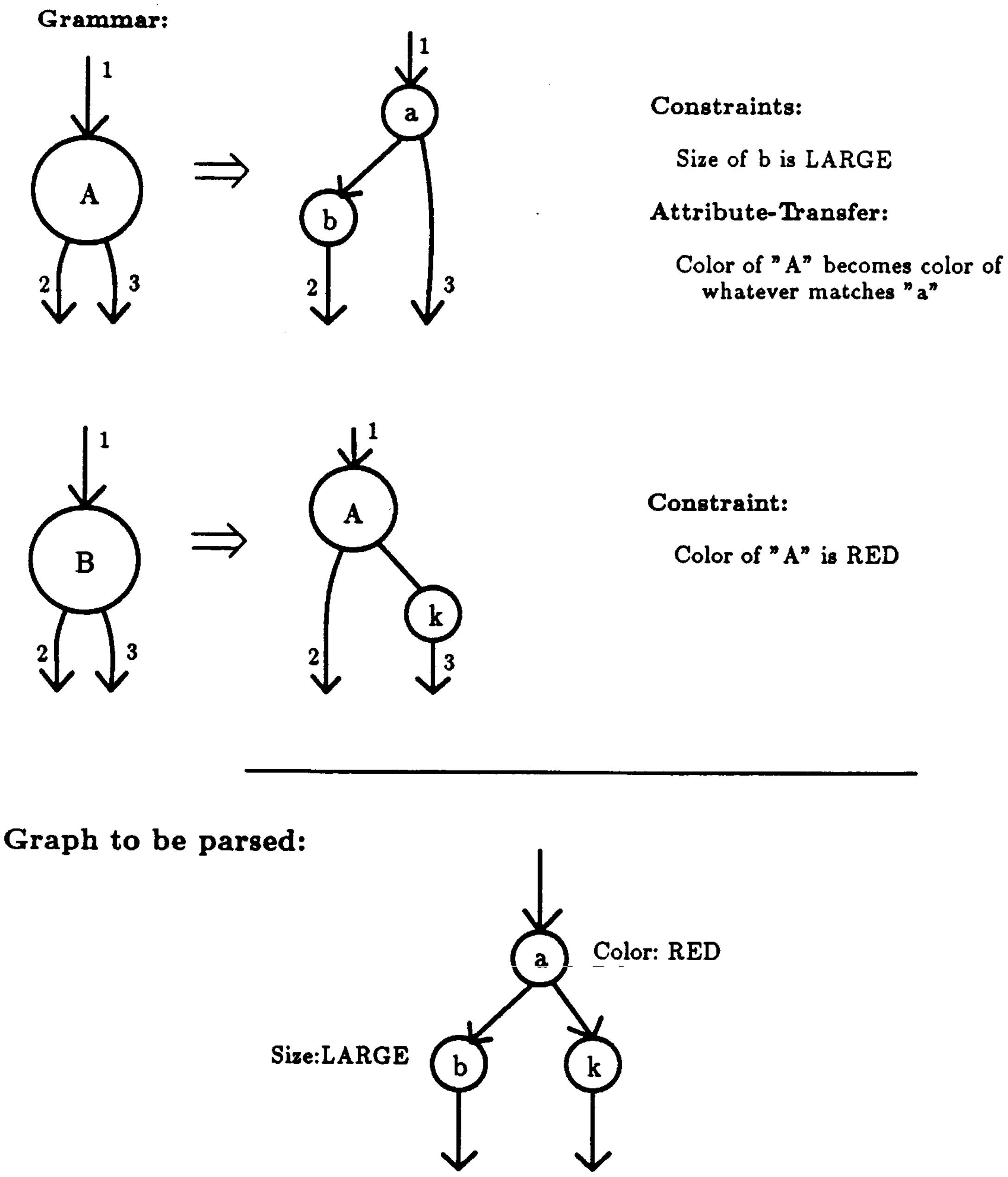


Figure 2.16: An Example Using Attributes, Constraints, and Attribute-Transfer

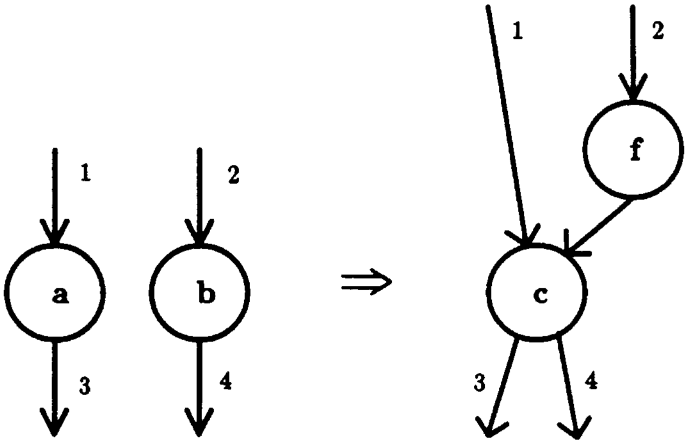


Figure 2.17: A Transformation Rule

are evaluated earlier than the rest. These constraints are evaluated as the nodes are scanned by the parser's read head. Other constraints are not evaluated until after the structural parsing of a rule is finished and a recognizer is ready to complete.

Transformation Rules

Often it is helpful to view the graph being parsed from more than one perspective, especially when a parse fails. A mechanism has been built on top of the parser which allows transformations to be made to the graph being parsed so that it may be seen in more than one way.

Each allowable transformation is specified in terms of a *transformation rule* which is context-sensitive. Its left-hand side specifies an alternate way to view the subgraph which matches its right-hand side. A sample transformation rule is shown in Figure 2.17. When the right-hand side of a transformation rule is recognized, it may be replaced by the rule's left-hand side. This means that Graph A in Figure 2.18 may be transformed into Graph B in Figure 2.18 by applying the transformation rule of Figure 2.17. Theoretically, this replacement will create a new input graph for each transformation rule that is applied, each showing a separate point of view of the transformed subgraph. However, for pragmatic reasons, the input graph is actually shared among the transformations, allowing the edges on the left fringe of the subgraph added to fan out from the input edges of the original subgraph and those

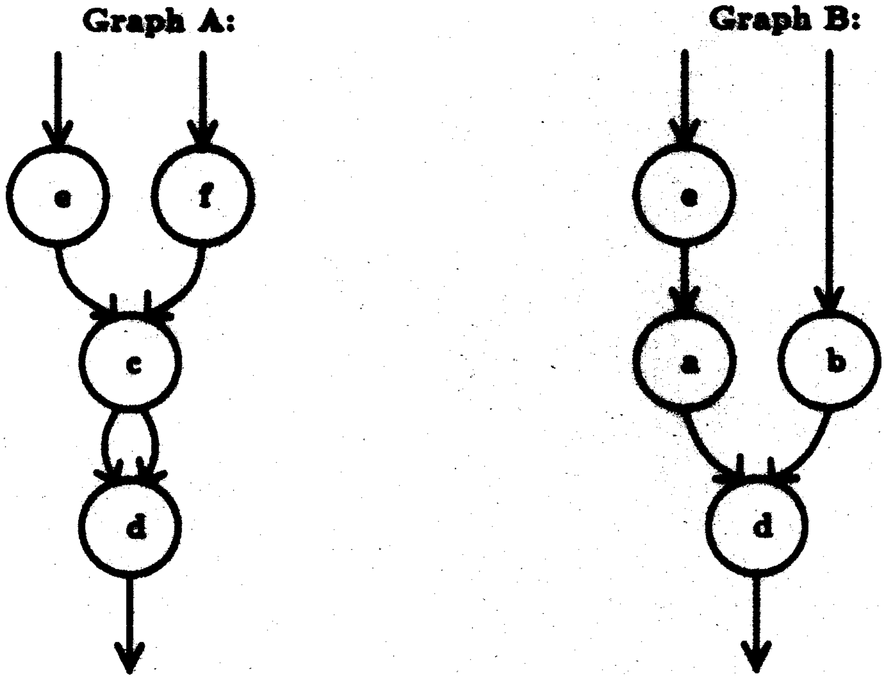


Figure 2.18: Graph A May Be Transformed into Graph B

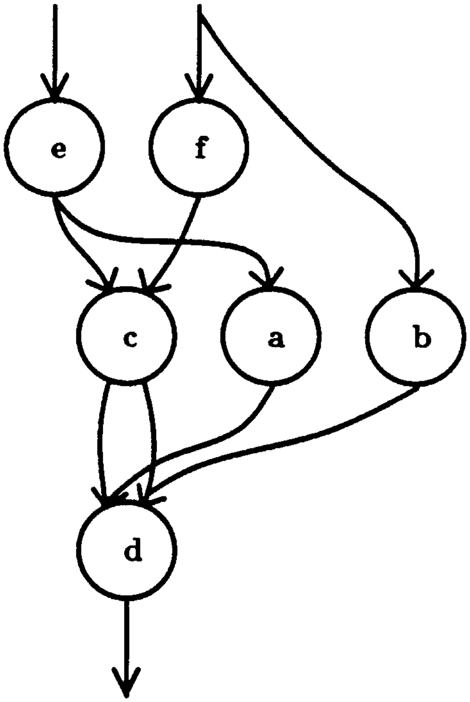


Figure 2.19: Result of Transforming Graph A

on the right fringe to all fan into the output edges of the original subgraph. This can be visualized as "sewing a patch" onto the input graph, where the transforming subgraph is the patch being placed over the subgraph that is to be seen in a different way. The actual result of transforming Graph A in Figure 2.18 is shown in Figure 2.19.

Since a transformation specifies two distinct ways to view a subgraph, the fan-in and fan-out of the transformed subgraph's fringes should not be treated as normal fan-in or fan-out by the parser. It would be an error for the graph in Figure 2.20 to be recognized as a subgraph of the graph in Figure 2.19. Unfortunately, in the current system, there is nothing to prevent a pattern from being matched that is partly in the original graph and partly in the transformed one. (This is because the implementation was not made robust enough due to time constraints on the project.) This is a weakness that must be eliminated if the transformation mechanism is to be used in a general way (as in the applications discussed in Chapter 3). For the current recognition system, however, the transformation facility is adequate since the Recognizer only uses the facility in a limited way to deal with arity mismatches between nodes. (This will be

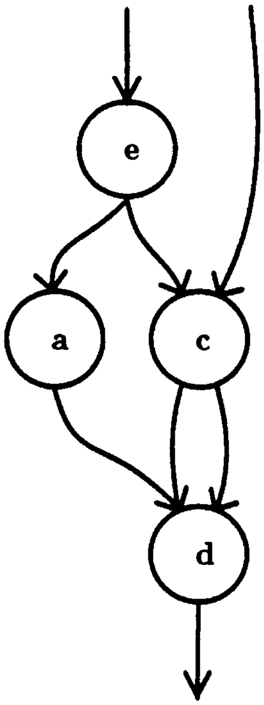


Figure 2.20: A Subgraph Which Shouldn't Be Found in the Transformed Graph A

discussed in sections 2.3.2 and 2.3.3.) Because the class of transformations is restricted and the clichés to be recognized do not contain any patterns that may use part of a transformed graph and part of the original graph, the problem of recognizing a pattern that is partly in both graphs does not arise.

There are a couple of other points to note about transformation rules. One is that transformation rules have attribute-transfer specifications just as normal rules do. These compute and transfer attributes from the graph matching the right-hand side to the left-hand side graph into which it is transformed.

Another point is that transformation rules are not used in regular parsing to transform the graph. Instead, the parser only attempts to recognize their right-hand sides in the graph. Rather than reducing the right-hand side to the left during parsing, the graph is transformed in a separate phase (the Transform phase, shown in Figure 1.1). After being transformed, the graph is reparsed in case the transformation allows previously failing parses to succeed.

There are many uses of the transformation mechanism. As will be seen, it is a way of introducing program transformations in a limited and controlled way. Other uses will be described in the future work section. Among these is a facility for performing recognition in buggy programs.

Restarting Suspended Parses

A third mechanism which is built on top of the parser enables failing parses to be temporarily suspended and then restarted where they left off, usually after some change has been made to the graph (via a transformation).

Because the parser is agenda-based, it is easy to suspend a parse temporarily. Instead of killing an item when its rule's right-hand side doesn't match the subgraph currently being scanned, the item is frozen along with its state, derivation lattice, and reason for failing so that it can be resuscitated after the input graph has been modified. When it is restarted, it picks up where it left off by reinstating its read head pointers in the input graph.

Selecting which parses to restart may be done on the basis of properties such as the location in the graph where they ran into trouble or the reason that they failed. This means, for example, that a transformation may be made and then only those parses which were parsing the section of graph affected by the transformation may be restarted. The Recognizer uses this approach.

The freezing and resuscitating technique is an optimization that allows the Recognizer to make only one parsing pass which may be interrupted by transformations. During this parsing pass, several things are done in parallel: normal recognition of clichés occurs; transformable sections of the graph are recognized; parses that cannot be carried through to completion are suspended until the subgraph they are parsing is modified. After a transformation, parsing resumes, allowing suspended parses to complete if possible. This is much more efficient than having several parsing passes.

Other Uses of Transformations and the Restart Mechanism

Both the transformation and the restart mechanisms are flexible in that either may be employed at any time. In the current system, they are used hand-in-hand. However, either may be used independent of the other. For example, sometimes parses may fail because of a constraint violation. The parse may be restarted without performing any structural transformations on the graph. The only thing that is changed is that the violated constraint is assumed to hold true. Chapter 3 will describe in more detail the ways in which these mechanisms may be useful.

2.3 Using the Extended Parser to Parse Plans

There are several obstacles to performing program recognition by parsing. Some problems arise because the plan representation doesn't canonicalize control flow information enough to allow subgraph matching alone to be used in recognizing clichés. Another problem stems from the fact that the recognition of plans which use constants may require that the program be seen in more than one way. A third problem is that plans may contain cycles in data or control flow arcs. These are not allowed in flow graphs and therefore need to be converted to another form. All of these problems are overcome by using the mechanisms described in the last section. A fourth problem, not solved by these mechanisms stems from the fact that programs cannot always be entirely reduced to the single start node of a grammar. Parts of the program's plan may need to be ignored and information must be gathered from subparses. The next four sections will deal with each of these problems separately.

2.3.1 Subgraph Matching

The Recognizer finds those sections within the program's plan which match clichés in the library. Since it does this by parsing, it finds subgraphs of the program's graph which match the right-hand sides of rules in the grammar. A fundamental part of parsing graphs, then, is subgraph matching. For example, a simple straight-line cliché is the *Interval-Length* cliché which computes the absolute difference between two input numbers. Its plan is shown in Figure 2.21. The following (contrived) program contains the Interval-Length cliché.

```
(DEFUN MULT-LENGTH (X Y K)
  (* (ABS (- X Y)) K))
```

The plan for MULT-LENGTH is shown in Figure 2.22. It contains the Interval-Length plan as a subgraph. Once this subgraph reduces to the left-hand side of the grammar rule induced by the Interval-Length cliché, the code corresponding to this subgraph can be described as computing the length of the interval between X and Y.

Why Subgraph Matching Cannot Be Used Directly on Plans

Performing subgraph matching on plans does not always lead to the recognition of all the clichés which occur in a program. There are two major problems. Both stem from the fact

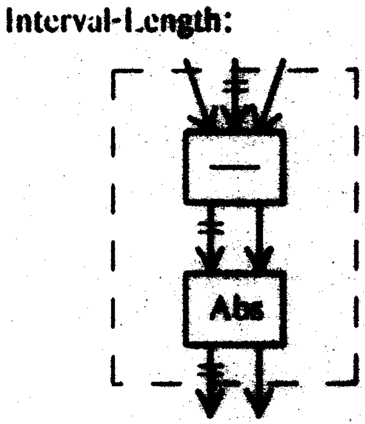


Figure 2.21: The Plan for the Interval-Length Cliché

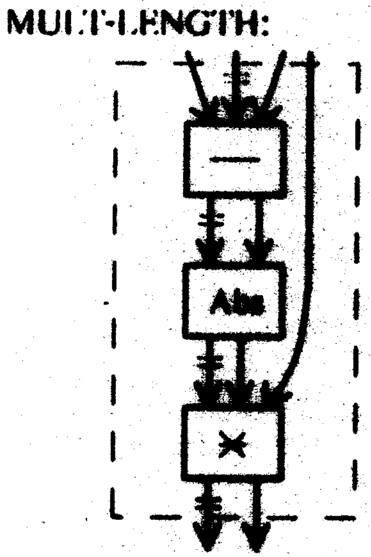


Figure 2.22: The Plan for MULT-LENGTH

that the Plan Calculus does not sufficiently canonicalize some aspects of the program related to control flow. In particular, even though it is able to abstract away from which control flow constructs are used in a program, it is too restrictive in representing the order of execution of operations and how joins merge data and control flow. The result is that two programs may be semantically equivalent but have different plans. This is a major obstacle to using subgraph matching in comparing two plans.

There are two problems with the Plan Calculus representation which hinder subgraph matching. First, it fails to capture the transitivity of control flow. Second, it doesn't treat joins as being associative.

As an illustration of the first problem, that transitivity of control flow is not represented, consider the graphs in Figure 2.23. The smaller graph to the right ("FH") represents a cliché which should be found in the plan to the left ("FGH") which is the plan for the following function.

```
(DEFUN FGH (X)
  (LET ((Z (F X))
        (Y (G X)))
    (I (H Z) Y)))
```

However, this cliché cannot be found because its plan definition does not occur as a subgraph of FGH's plan. There is a control flow arc missing between the **F** and **H** nodes. By transitivity of control flow, there is control flow between the two nodes, but the plan does not show it explicitly.

In order to solve the transitivity problem, the transitivity of control flow must be captured. One way to do this is to transitively close the entire plan with respect to control flow. This is an expensive computation, however. The extra control flow arcs bring added complexity to the graph. Furthermore, the transitive closure computation is made more complicated by sections of the plans which contain splits and joins and which need to be treated specially. In particular, all sections of a plan which have splits and joins at either end must be treated as a single node when closing the rest of the graph. Within the split-join section, each branch of the split must be transitively closed independently. This computation is combinatorially explosive.

The second major problem with the Plan Calculus is that it doesn't treat joins as being associative. Joins, when nested, can associate in a variety of ways. For example, both of the

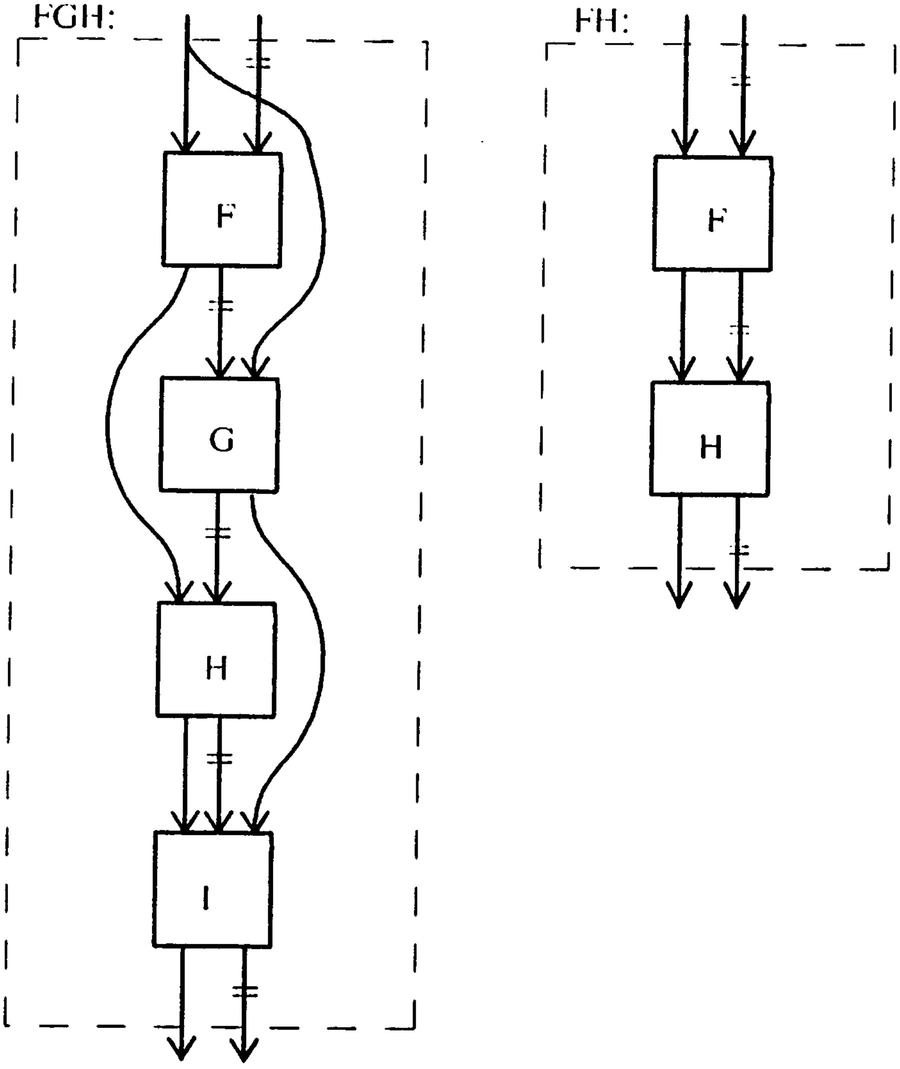


Figure 2.23: The Plans for the Program FGH and the Cliché FH

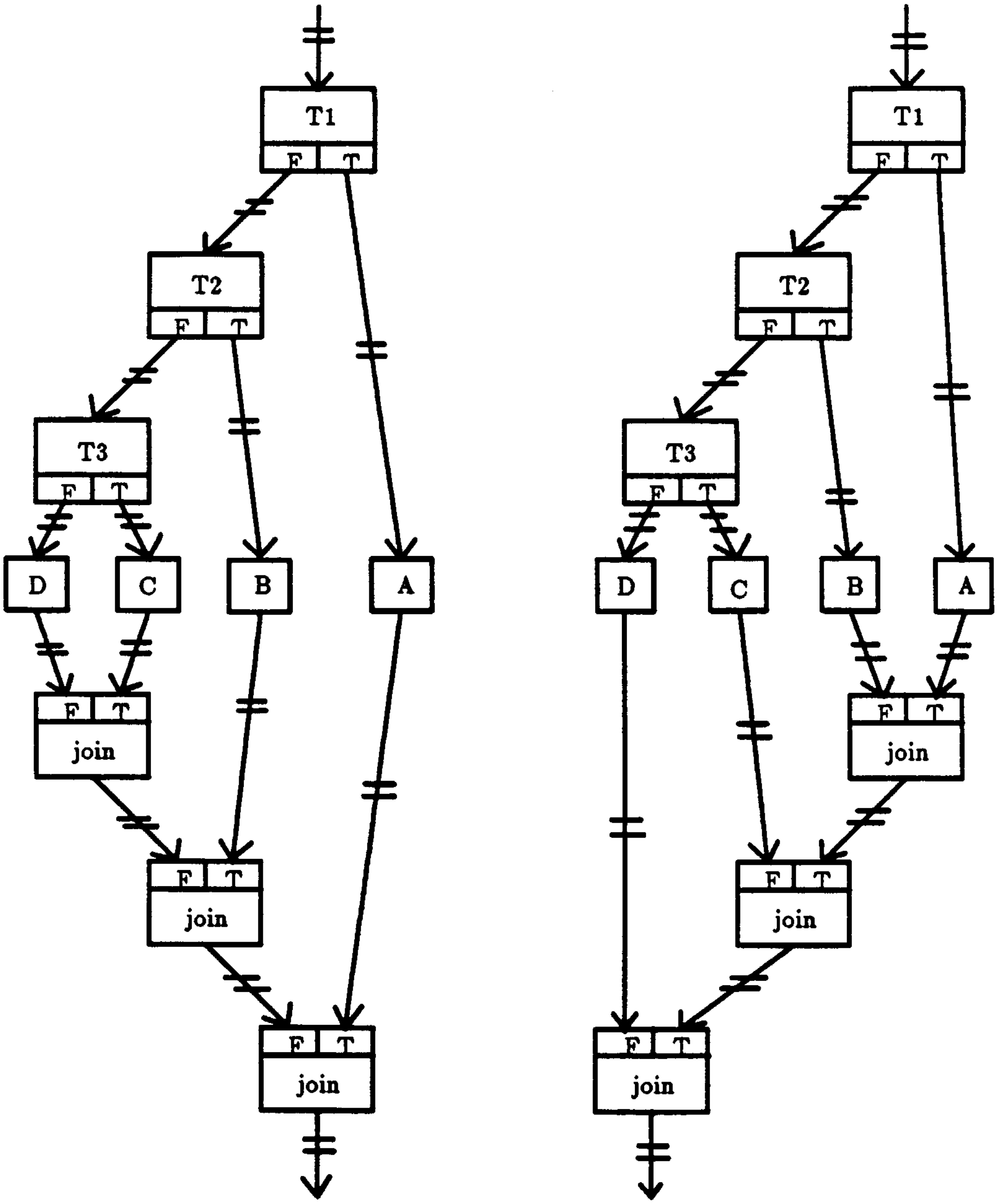


Figure 2.24: Two Equivalent Plans in Which Joins Associate Differently

plans shown in Figure 2.24 are semantically equivalent, since joins are associative. Yet they do not structurally match each other.

Expressing Transitivity of Control Flow

Rather than representing control flow as arcs and explicitly closing them under transitivity, the Recognizer's approach is to remove control flow arcs from the plan altogether and represent the information that they contain in attributes on nodes.

Control flow arcs specify two types of information which need to be either recorded in attributes or thrown away if not needed for recognition. One is the order of execution of operations chosen by the programmer. This order is irrelevant to the recognition process, since the only order between the operations which must be the same in two equivalent plans is that imposed by data dependencies. Data flow arcs provide this structural constraint. For example, FGH1 and FGH2 in Figure 2.25 are semantically equivalent even though the order in which **F** and **G** are executed is not the same. (This is because the Recognizer assumes that **F** and **G** do not have any side effects. In the future, if the Recognizer is to analyze programs which have side effects, the data flow arcs must model the side effects.) This type of information is therefore not recorded.

```
(DEFUN FGH1 (X)                (DEFUN FGH2 (X)
  (LET ((Z (F X))                (LET ((Y (G X))
        (Y (G X)))                     (Z (F X)))
    (I (H Z) Y)))                  (I (H Z) Y)))
```

Figure 2.25: Two Semantically Equivalent Programs

The second type of information which control flow arcs provide is relevant to the recognition process and is therefore converted to attributes. This information specifies how to group together operations which are in the same *control environment*. Control environments specify under which conditions operations are performed. Each operation contained in a control environment is performed the same number of times as every other operation in that control environment. That is, all operations in a control environment *co-occur*. All operations in a straight-line expression are in the same control environment. When control flow splits, two new control environments are created, one for the true side and one for the false side of the split. Each branch of the split is in a separate control environment.

Each node in a plan has an attribute which tells the control environment of the operation it represents. Splits are different than nodes that represent operations in that they have two extra attributes, *success-ce* and *failure-ce* which specify the control environments of the two sides of the conditional.

Control environments form a partial order in which the order relation is called $\sqsubseteq$. A control environment, ce_i, is less than or equal to another control environment, ce_j, if operations in ce_i are performed at least as many times as those in ce_j. For example, in the plan in Figure 2.26 (shown without data flow arcs for clarity), all operations are annotated with a control environment (in the form "*ce*" subscripted with a unique number). In the figure, $ce_4 \sqsubseteq ce_2$ because the operation **C** in ce_4 is only performed when the test **B** fails. Control environments of opposite branches of a split, such as ce_3 and ce_4, are *incomparable.*

It is also useful to define a partial "addition" operation on control environments. The sum of the number of times operations are performed on each branch of a conditional is equal to the number of times the operations surrounding the conditional are performed. That is, the sum of two control environments gives the control environment of the split which created them (as well as the control environment of any join which merges them). An addition operation is therefore defined as: given two control environments, ce_l and ce_m, $ce_l + ce_m = ce_n$ iff ce_n is the control environment of the split which created ce_l and ce_m. In figure 2.26, $ce_4 + ce_5 = ce_2$. That is, the sum of the number of times operation **C** is performed and the number of times **D** is performed in any execution of the program is equal to the number of times operation **B** is performed.

The addition operation is commutative and associative. It is a partial function since it is not always defined. The addition of two control environments is only defined when the control environments are on opposite sides of the same split. For instance, $ce_1 + ce_2$ is undefined.

Given this definition of addition, subtraction may be defined as: if $ce_l + ce_m = ce_n$ then $ce_n - ce_m = ce_l$, and $ce_n - ce_l = ce_m$. In figure 2.26, $ce_1 - ce_2 = ce_3$.

Constraints on control environment attributes may be used to solve the control flow tran-

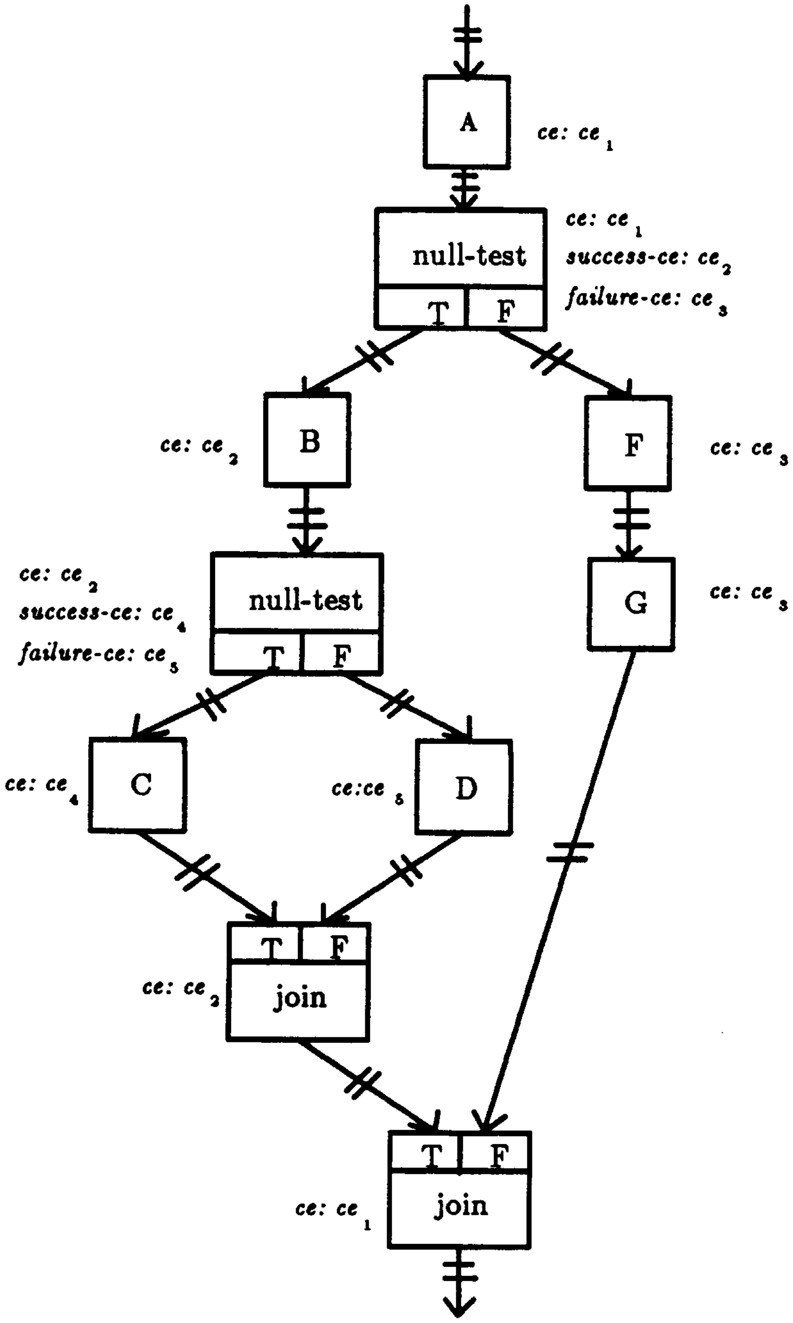
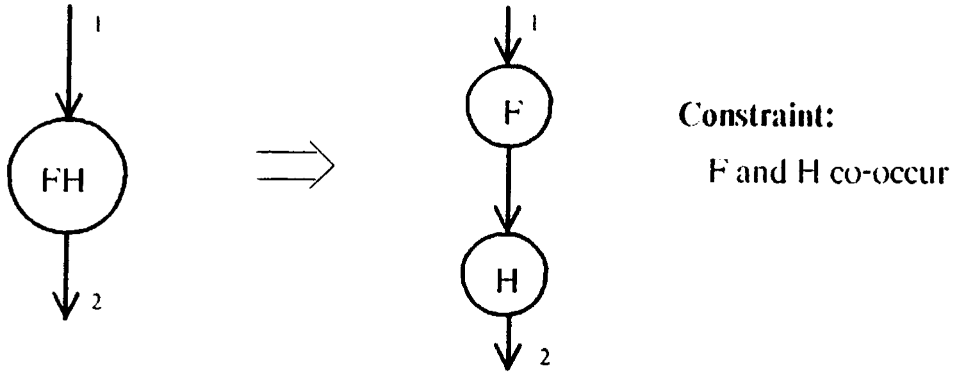


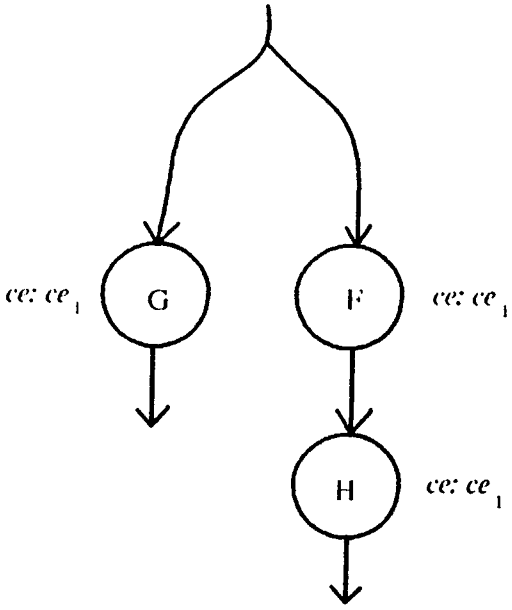
Figure 2.26: Control Environments

sitivity problem. In the FGH example, the grammar rule induced by the FH cliché becomes:



This rule places a constraint on the control environment attributes of the nodes **F** and **H**, requiring that they have the same control environment, rather than requiring that **F** and **H** have a control flow arc between them. This means that instead of requiring that **H** be executed immediately after **F**, the rule requires only that **H** is executed whenever **F** is executed and that **H** always receives data flow from **F**.

The flow graph projection of the plan for FGH is:



The control flow arcs have been removed and control flow attributes have been placed on the nodes. The right-hand side of the rule for FH would be found in this graph and the

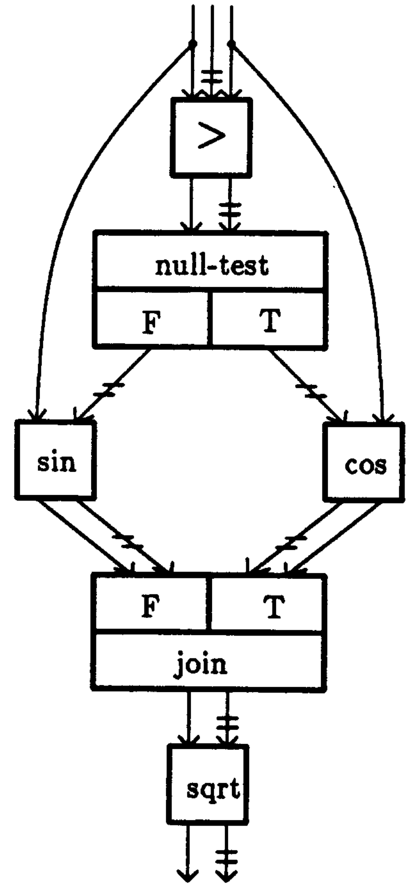


Figure 2.27: A Sample Plan Containing a Join

constraint that F and H be in the same control environment would be satisfied. Thus, the cliché FH would be recognized in FGH.

Expressing Associativity of Joins

The problem of representing the associativity of joins is also solved by removing information from the plan and placing it in attributes. In particular, all joins are removed from the plan, causing all data flow arcs merged by the join to fan into operations which received the merged data flow from the join. Thus, no particular way of associating the joins (e.g., always associating them to the left) needs to be enforced in order to canonicalize the graph.

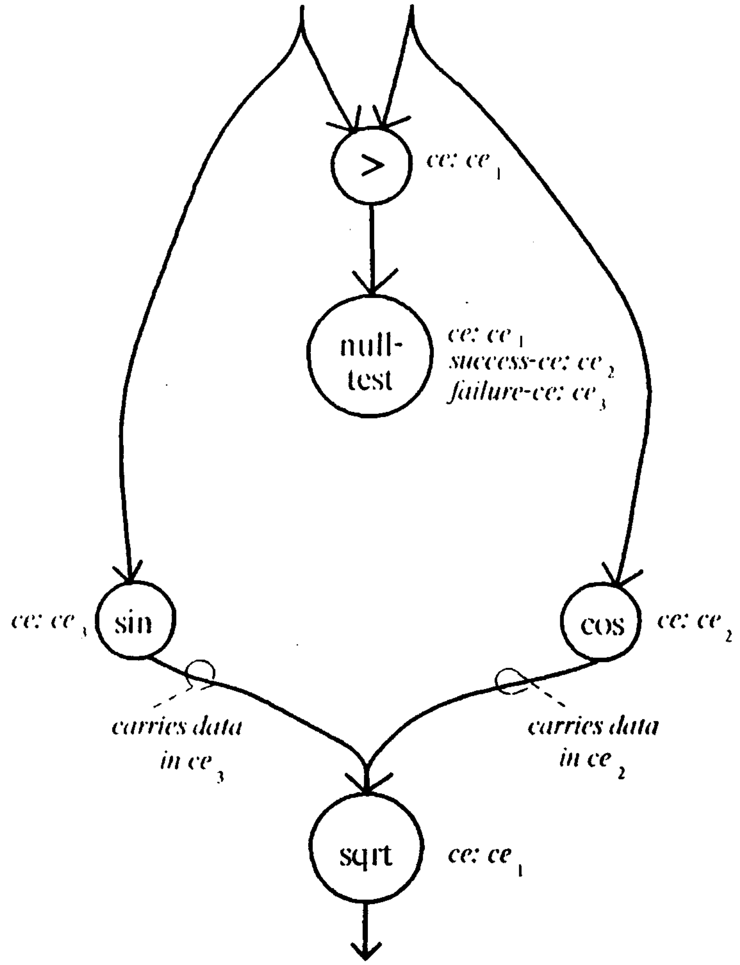
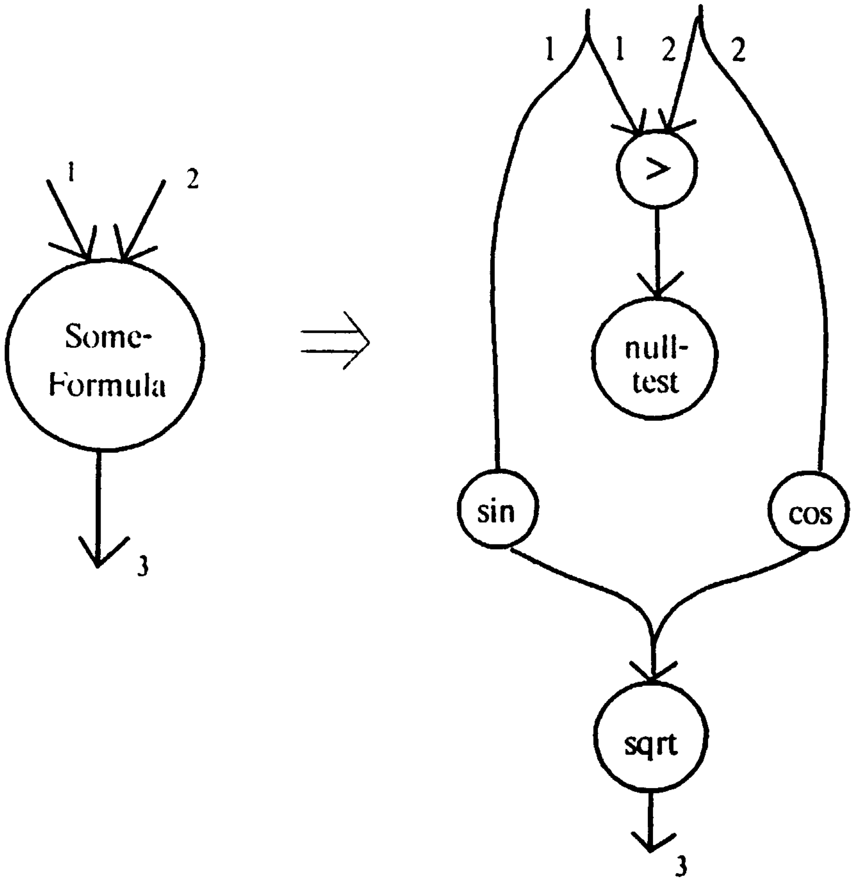


Figure 2.28: An Attributed Flow Graph Projection

The information that needs to be converted to attributes is: for each incoming data flow arc, in which control environment is data being carried by that particular arc? For example, suppose there were a plan definition such as the one shown in Figure 2.27. The join in Figure 2.27 specifies that when the **true** side of the split is taken, SQRT will receive data flow from the COS node and when the **false** side is taken, it will receive data flow from SIN. This information is converted into attributes on the edges of the graph which are being merged. The attribute on each edge tells in which control environment the edge carries data flow. For instance, the plan in Figure 2.27 becomes the annotated flow graph in Figure 2.28 when the joins and control flow arcs are removed.

The typical constraint placed on an edge in a grammar rule is that it must carry data



Constraints:

1. Data flows into "sqrt" from "sin" in the failure-ce of "null-test"
2. Data flows into "sqrt" from "cos" in the success-ce of "null-test"

Figure 2.29: A Grammar Rule with Data Flow Constraints

```
A.     (SQRT (COND ((> X Y)
                    (SIN X))
                   (T (COS Y))))


B.     (LET ((SIN-X (SIN X)))
         (SQRT (COND ((> X Y) SIN-X)
                     (T (COS Y)))))
```

Figure 2.30: Code Fragments in Which Some-Formula Appears

flow in a particular control environment. For example, the grammar rule induced by the *Some-Formula* plan is shown in Figure 2.29. (The constraints are given informally so that the reader may understand them without having to understand the syntax of how they are specified. Their actual syntax and definitions are given in Appendix A.)

The Some-Formula plan will be recognized in both code fragments in Figure 2.30, even though in fragment B, SIN is not in the failure-ce of the null-test. This is because Some-Formula's constraint is not on the attributes of the SIN node but rather on the edge between SIN and SQRT.

In summary, determining whether two plans are semantically equivalent cannot be done simply by matching them against each other as graphs. Some of the information in plans must be converted to annotations on the plan's nodes and edges. What is left is an attributed flow graph projection of the plan. Two plans are equivalent if their flow graph projections match and their attributes satisfy any constraints that either plan has on the attributes of the other. Therefore, the core operations of recognition via parsing are subgraph matching (of flow graphs) and logical subsumption (of attribute values).

How Extensions to the Parser Enable Parsing Flow Graph Projections

It can now be seen why extensions were made to the parser to allow it to parse graphs containing sinks and fan-in edges, as well as fan-out and straight-through edges. When control flow arcs are removed, split nodes become sinks while the removal of joins causes data flow arcs to fan-in. These extensions are all necessary, for example, in recognizing the *Absolute-Value* cliche in the following code.

```
(DEFUN ABS-VAL (X)
  (COND ((PLUSP X) X)
        (T (NEGATE X))))
```

The flow graph projection of the plan for ABS-VAL is shown in Figure 2.31. The null-test is a sink. There is a straight-through edge from the input to the output, representing the fact that the input to absolute value is given as the output without being changed if the input is positive. There is fan-out of data flow, indicating that the input data may be used by NEGATE or simply returned. The fan-in of data flow indicates that the output of Absolute-Value may come either from the NEGATE operation or directly from the input.

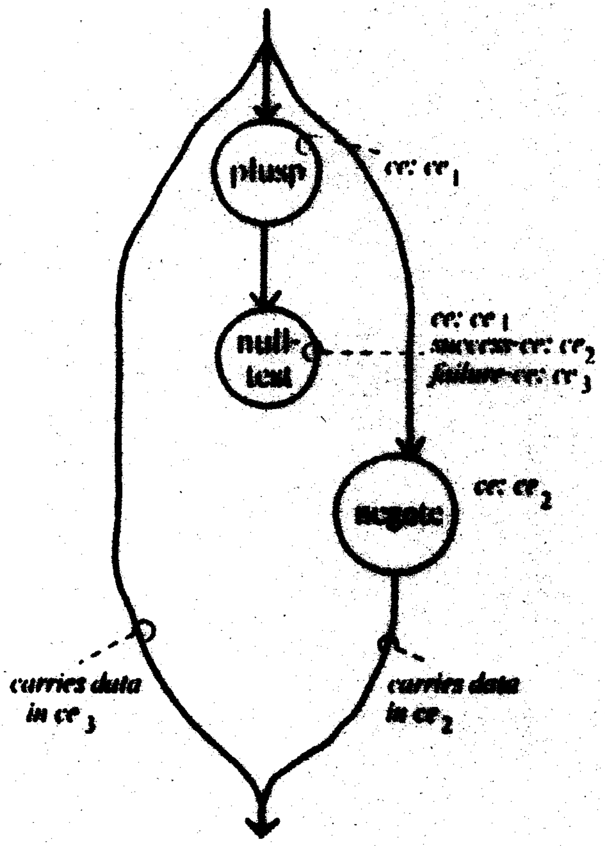


Figure 2.31: The Flow Graph Projection of the Plan for ABS-VAL

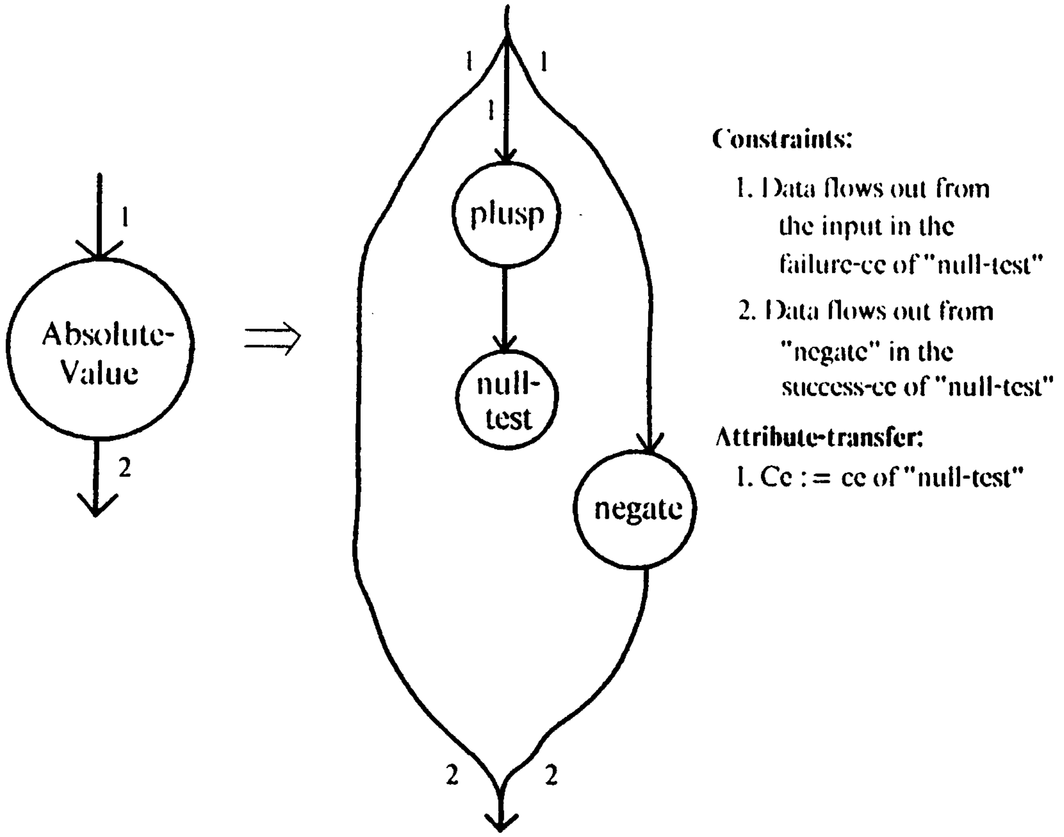

1
Absolute-Value
2
⇒
1
1
1
plusp
null-test
negate
2
2
Constraints:
1. Data flows out from the input in the failure-cc of "null-test"
2. Data flows out from "negate" in the success-cc of "null-test"
Attribute-transfer:
1. Cc := cc of "null-test"

Figure 2.32: The Rule for the Absolute-Value Cliché

The rule induced by the Absolute-Value cliché is shown in Figure 2.32. (The formal definition of this rule and the rest of the grammar used by the Recognizer is given in Appendix B.) In this rule, the mapping from the input port of the left-hand side node to all leading edges requires that the input to both NEGATE and PLUSP, and the output (in the case of positive input) all come from the same place. Likewise, the mapping from the left-hand side node's output port to both trailing edges requires that the data flow from NEGATE and from the input must be used in the same place.

This section has shown how the extensions made to the parser and the attribute mechanism are used to solve one of the difficulties of performing recognition by parsing. The next difficulty that will be discussed is the problem of recognizing clichés in programs when the clichés and programs contain constants.

2.3.2 Dealing with Constants

When a program or cliché uses a constant, it may represent the use of the constant in two different ways: either as an explicit input into the operation which uses the constant, or as being incorporated into the definition of the operation. For example, two graphs for incrementing an input integer are shown in Figure 2.33. (In Figure 2.33a, the fact that the second input into the "+" is coming from a constant whose value is 1 is represented in the flow graph as an attribute on the input arc. This is drawn as a hooked line around the input arc with a 1 beside it.) In Figure 2.33a, the constant is represented explicitly, while in part b of the figure, the constant is implicit in the "1+" operation.

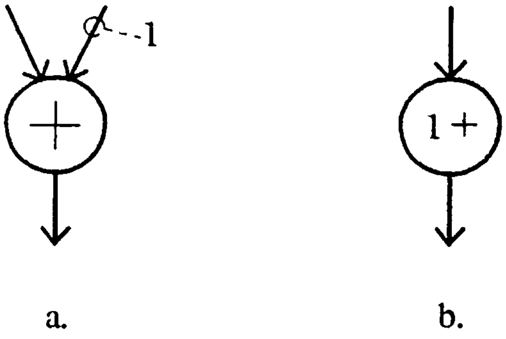


Figure 2.33: Two Graphs for Incrementing an Integer

Constants in Grammar Rules

When a cliché containing a constant is translated into a grammar rule, the constant cannot be represented explicitly. If it were, the modularity of the grammar would be violated. For example, the cliché *Average* is computed by the expression (/ (+ X Y) 2). *Average* should be thought of as having two inputs — the numbers to be averaged. It does not have an extra input for the constant 2 (as is shown in the incorrect rule in Figure 2.34).[2] Furthermore, allowing *Average* to have an additional input would destroy the modularity of the grammar since the extra input must be propagated up through all rules that use *Average*, as in the

[2] The "X"s on either side of the rule indicate that it is an incorrect rule.

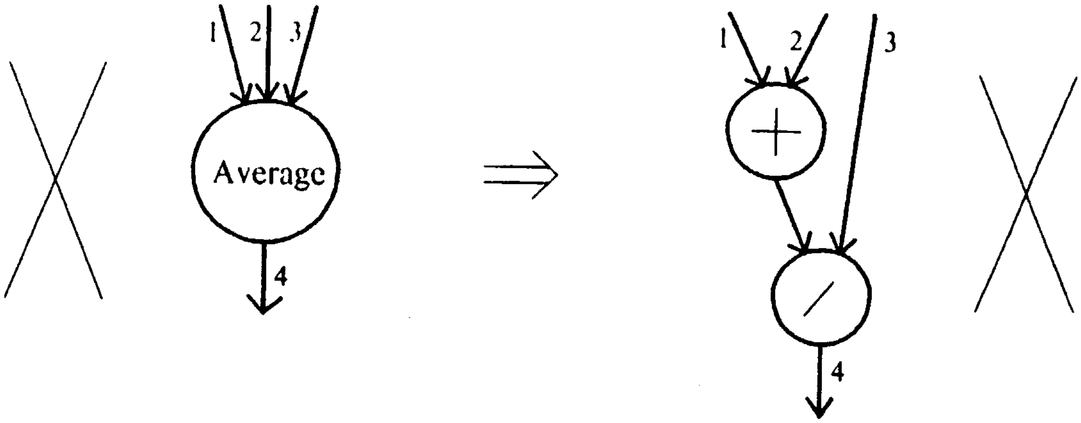


Figure 2.34: An Incorrect Grammar Rule for the Cliché *Average*

(incorrect) rule shown in Figure 2.35.[3]

This problem also cannot be solved by having an input from the constant on the right-hand side but not on the left-hand side (as in Figure 2.36) because it violates the constraint on grammar rules that their left- and right-hand sides must have the same arity.

The way the Recognizer deals with this problem is that when an operation in a cliché receives data flow from a constant, the operation is transformed into the equivalent partially evaluated function which results from closing the function with respect to the constant input. (This is currying if the constant is the first argument to the function.) Thus, when the plan becomes a grammar rule, the rule's right-hand side has no constants. For example, when one of the inputs to "+" is constrained to always be 1 in a cliché, the operation is transformed into "1+" when the cliché is translated into a grammar rule. This means there is only one rule for the plan for incrementing an integer (i.e., Figure 2.37).

Incorporating constants into operations in a rule's right-hand side must be done recursively. If a constant is incorporated into a unary function, yielding a new constant, then that constant must be incorporated into the operations that use it. For example, consider the graph in Figure 2.38(a) to be some rule's right-hand side. If the constant 2 is incorporated into the operation SQRT, then a new constant is formed which represents the constant $\sqrt{2}$ (Figure 2.38(b)). This constant must be recursively incorporated into the * operation as in Figure 2.38(c).

[3] The squiggles in the rule indicate that there is more to the right-hand side graph without going into its details.

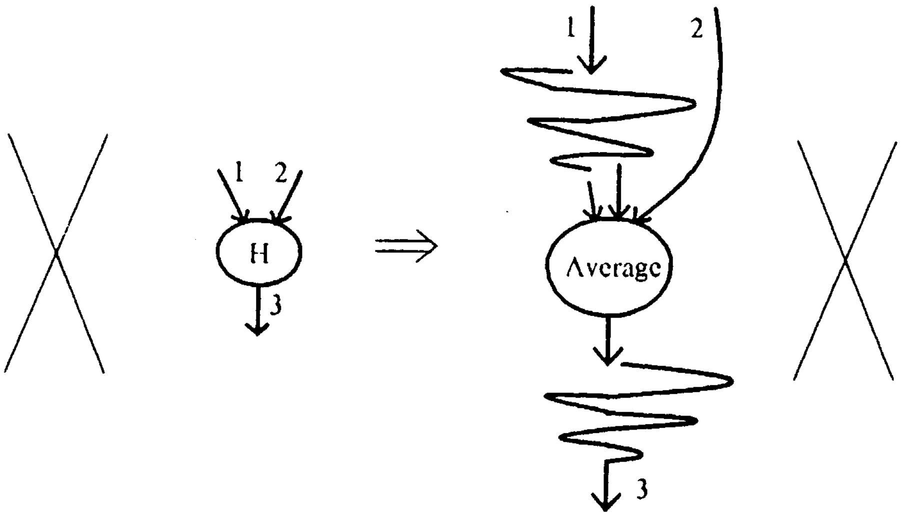


Figure 2.35: A Rule Exhibiting Bad Modularity

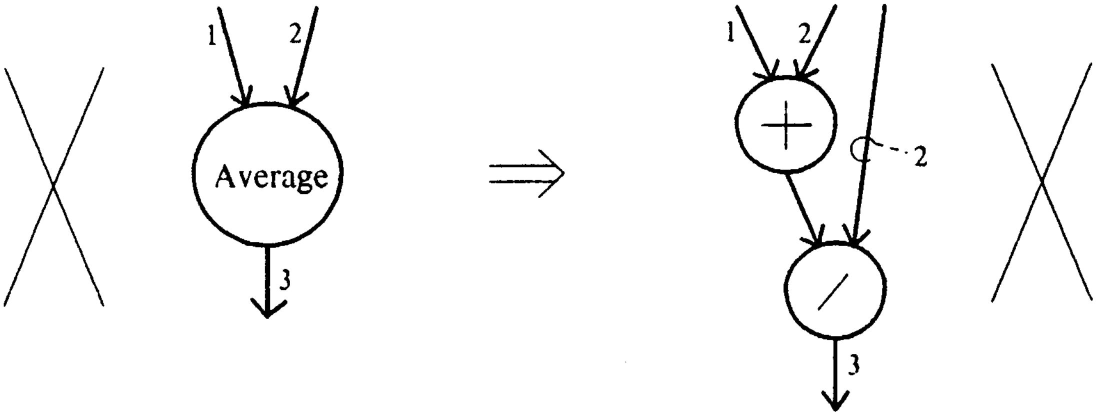


Figure 2.36: An Incorrect Rule Violating the Arity Constraint

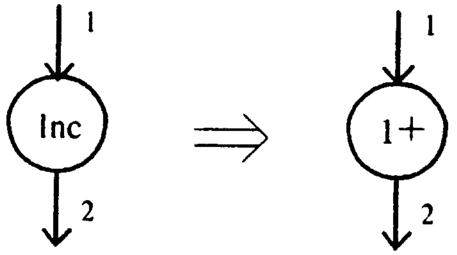


Figure 2.37: The Rule for Incrementing an Integer

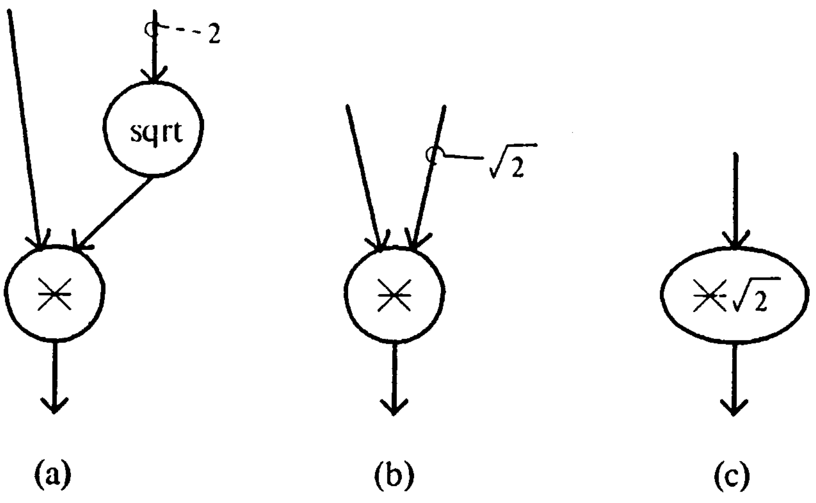


Figure 2.38: Recursively Incorporating a Constant into Operations

Later in this report, some rules which contain constants on their right-hand side may be shown with the constant represented explicitly as in Figure 2.36. This is done for clarity. The reader should assume in these cases that in the actual rules used by the Recognizer, the constants are incorporated into the functions that use them.

Constants in Programs

Because programs may represent the use of a constant in two different ways (either as an explicit input or as being incorporated into the operation), two difficulties arise. First, since constants are always represented implicitly in grammar rules, a rule's right-hand side won't match the graph of a program which uses the constant explicitly. For example, the right-hand side of the rule for *Increment* given in Figure 2.39a will not match the graph for (+ X 1) shown in Figure 2.39b.

The second difficulty is that if a program represents a constant as being incorporated into some operation, then it will not match with a rule's right-hand side in which the operation does not require a constant as input. For example, the right-hand side of the rule for *Add* in Figure 2.40a will not be recognized in the graph for (1+ X) in Figure 2.40b.

The way that the Recognizer deals with the first problem, i.e., that clichés which use constants will not be recognized in programs which use constants explicitly (e.g., (1+ X) will

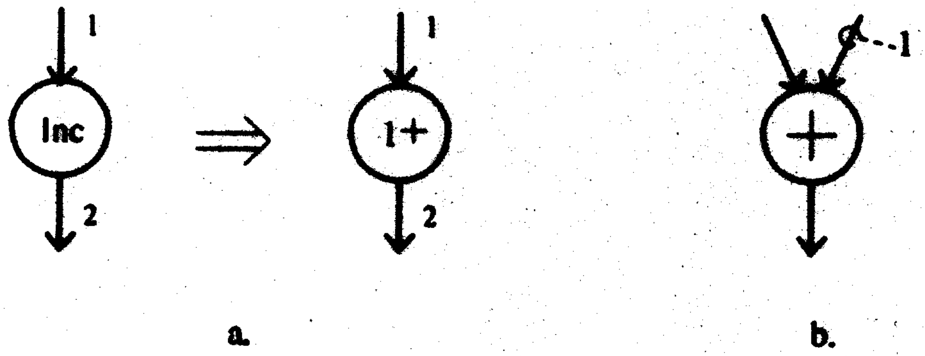


Figure 2.39: The Rule for *Increment* and a Program Graph

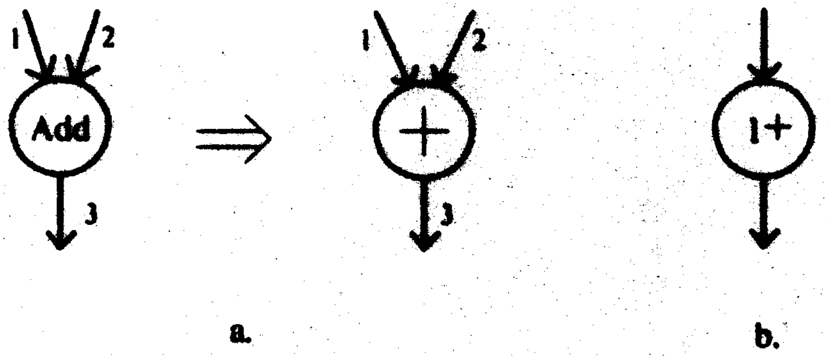


Figure 2.40: The Rule for *Add* and a Program Graph

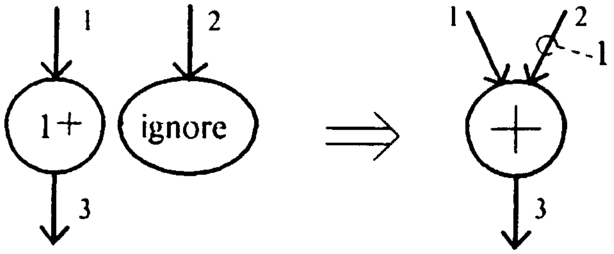


Figure 2.41: Transformation Rule for Viewing Constant in More Than One Way

not be recognized in (+ X 1)), is by allowing the program to be viewed in more than one way at once. The two ways that (+ X 1) may be viewed are those shown in Figure 2.33. This is achieved by using transformations along with the parse restart mechanism. The transformation rule needed is one which will transform representations of operations which receive data flow from a constant to the representation in which the operation incorporates the constant into its definition. For instance, the transformation rule needed to view (+ X 1) as the two graphs in Figure 2.33 is shown in Figure 2.41. This rule specifies that whenever a "+" operation occurs in a program and receives data flow from the constant 1 as its second argument, then the "+" may be transformed into the graph on the left-hand side of the rule. (The node of type *ignore* indicates that the input from the constant should be ignored. When the parser scans this edge, nothing needs to match with it.)

Once a transformation has been made, any parses which depended on seeing the graph in an alternative way are resuscitated. In the case of (+ X 1), the parse for recognizing *Increment* cannot succeed at first and is temporarily suspended. The right-hand side of the transformation rule of Figure 2.41 is found and during the Transform phase, the "+" is transformed into "1+". This allows the parse of *Increment* to complete successfully.

The second difficulty (i.e., that the plan for (+ X Y) will not be recognized in (1+ X)) is dealt with by always expanding the input form in which a constant is incorporated into an operation into the canonical form in which the constant is given as an explicit input. (This is done in the analysis phase of the recognition process.) This means that whenever (1+ X) is encountered, it is translated into (+ X 1), allowing the plan for (+ X Y) to be recognized in it. The expansion of forms in which constants are implicit in the operations that use them to forms in which they are explicit is always performed. The Recognizer relies on using trasformation rules to recognize clichés which contain constants and which therefore induce rules in which constants are implicit in the operations that use them. For example, when recognising the plan for *Increment* (Figure 2.40a) in (1+ X), the Recognizer canonicalizes (1+ X) to (+ X 1) and then uses the transformation rule of Figure 2.41 to view the "+" as a "1+".

```
(DEFUN POS-ELEMENTS (L)
  (LET ((NEW-L NIL)
        (E NIL))
    (LOOP DO
      (COND ((NULL L) (RETURN NEW-L)))
      (SETQ E (CAR L))
      (IF (PLUSP E)
          (SETQ NEW-L (CONS E NEW-L)))
      (SETQ L (CDR L)))))
```

Figure 2.42: A Function Containing a Loop

2.3.3 Loops

The desired analysis for the program in Figure 2.42 is that it enumerates the elements of the list L, filters out all positive elements, and accumulates them in the list NEW-L. This type of analysis requires that the loop be thought of as a straight-line composition of operations which act on sequences. That is, the loop must be *temporally abstracted.* Temporal abstraction is a representation technique, developed by Waters [42], Rich [28], and Shrobe [31], to capture the commonality between different loops. Each operation is seen as a *temporal fragment* acting on each element of a series of values, called the *temporal sequence*, simultaneously. For example, in POS-ELEMENTS, two of the sequences being operated upon are the series of values of L being given to CAR and the series of values of NEW-L being given to CONS. The CAR operation is viewed as a temporal fragment, called a *Map*, which takes a sequence of lists and outputs the sequence consisting of their CARs.

By analyzing a subset of the programs in the IBM Scientific Subroutine Package by hand, Waters [42] found that most loops can be temporally abstracted into a small number of basic loop fragments. Rich formally defines the basic loop plans for temporal fragments in [28].

This section shows how the Recognizer temporally abstracts loops by using grammar rules induced by the basic loop plans. First, it discusses the cyclic plan representation of loops and tells how the plans are translated into attributed flow graphs. Then, the basic loop plans will be described in more detail, showing how they become rules which serve to raise the cyclic view of the program up to a temporally abstracted, non-cyclic view.

Representing Loops

Loops may be represented in the Plan Calculus as either cycles in control and data flow arcs or as tail recursions. The Recognizer uses the cyclic plan representation of loops. When converting the plan to a flow graph, the cycles are broken and the graph corresponding to the loop body is parsed. The decision to represent loops as cycles rather than as tail recursions came from the realization that analyzing loops is the first step in being able to analyze recursive programs.

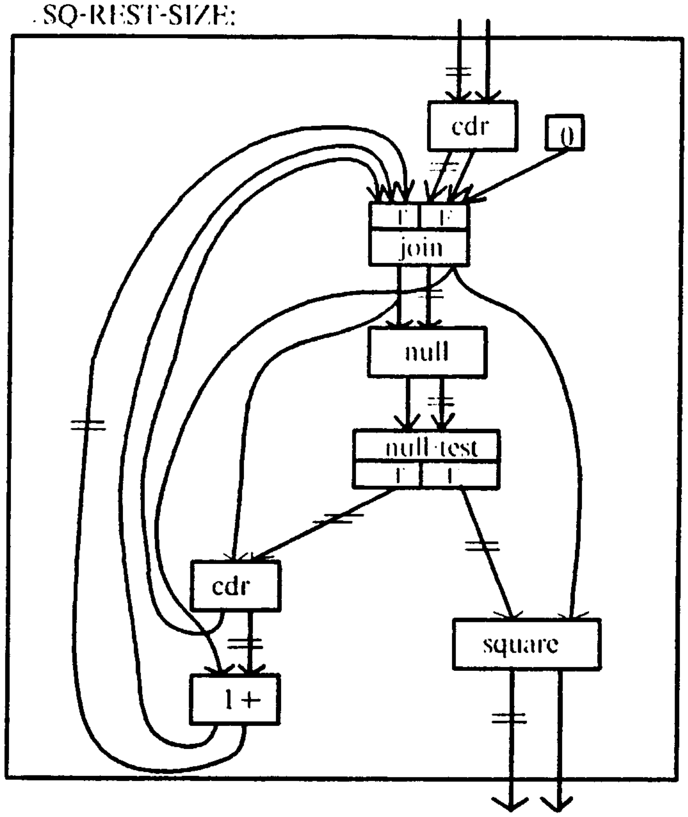
Difficulties arise in using either representation for loops. Cycles cannot occur in flow graphs, on the one hand. On the other, the use of recursive nodes would introduce additional problems having to do with arity mismatches between recursive nodes in grammar rules and those in the graph being parsed. (This will be explained in greater detail in the section on recursion in Chapter 3.) However, breaking the feedback arcs and parsing the loop body is easier than dealing with the arity mismatches. Furthermore, the information that the feedback arcs contain and that is converted to attributes is the same information that must be extracted from a tail recursive plan when it is translated to a flow graph. (The next section will describe what information is converted to attributes. This corresponds to the information that is extracted from recursive plans, as will be seen in the section on recursion.)

Because the information that must be converted to attributes is the same in both the cyclic and tail recursive representations and because using recursive nodes introduces more complexity, analyzing loops (as cycles) is a subproblem of analyzing recursion. The insights gained and the work done in analyzing loops form a basis for further research into recognizing clichés in recursive programs. Therefore, it was decided to solve the problem of recursion incrementally, rather than all at once.

Converting Loop Control Flow Information to Attributes

Figure 2.43a shows the plan for the contrived program in Figure 2.43b which computes the length of the CDR of a list and squares it. In order to parse this plan, the Recognizer must remove the cycle in control and data flow. It must then parse the flow graph projection of the body.

The information contained in the feedback arcs is converted to attributes in the following way. When a data flow feedback arc is broken, the ports which were the endpoints of the arcs



a.

```
(DEFUN SQ-REST-SIZE (L)
  (LET* ((REST (CDR L))
         (LENGTH 0))
    (LOOP DO
      (COND ((NULL REST) (RETURN)))
      (SETQ REST (CDR REST))
      (SETQ LENGTH (1+ LENGTH)))
    (SQUARE LENGTH)))
```

b.

Figure 2.43: The Plan and Code for SQ-REST-SIZE

are annotated with the attribute that the source port *feeds-back* to the sink port.

In addition to feedback correspondences, control environment information is also converted to attributes. The essential control flow information associated with loops can be summed up in three control environments. These are:

- **feedback-ce** — the control environment in which the control of a loop feeds back to the beginning of the loop.
- **outside-ce** — the control environment from which a loop is entered and into which it exits.
- **loop-body-ce** — the highest control environment of a loop's body. It is the control environment of the first operation reached in control flow when the loop is entered.

Figure 2.44 shows the plan for SQ-REST-SIZE on the left and the flow graph into which the plan is translated on the right. The flow graph is annotated with the appropriate attributes. "Feeds-back" attributes are denoted by subscripted asterisks. For example, the output of the CDR within the loop's body feeds back to the input of NULL and the input of itself. The control environments of the loop are attributes of the loop itself. The graph records a set of the important loop control environments for each loop. The outside-ce of the loop in SQ-REST-SIZE is ce_1, the loop-body-ce is ce_2, and the loop's feedback-ce is ce_3.

Grammar Rules for Loop Plans

Now that the flow graph representation for loops has been described, the grammar rules used to parse it will be explained. These rules are induced by the basic loop plans used to temporally abstract programs which contain loops.

Generation

The *Generation* plan takes as input a data object and a function and outputs a temporal sequence. The output sequence is the result of repeatedly applying the operation to the output of the preceding application of that operation. For example, a *Cdr-Generator* takes a list and the function CDR and produces a sequence consisting of all the successive sublists of the list. In the function POS-ELEMENTS, successive values of L are generated in this way. Similarly,

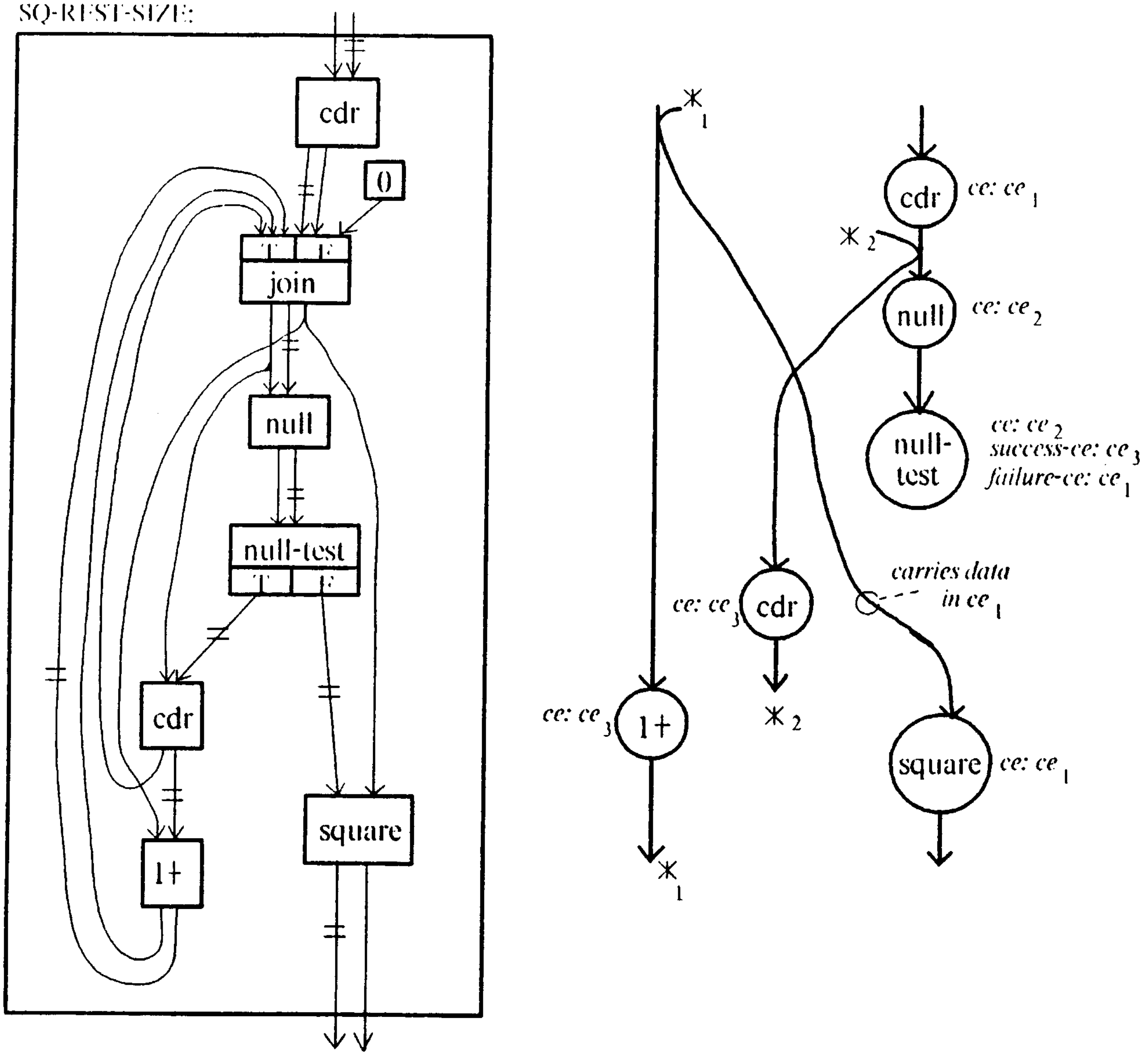
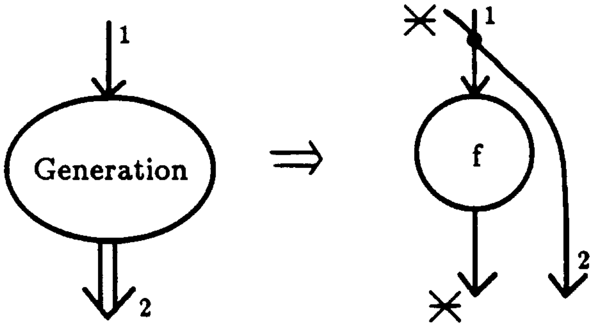


Figure 2.44: Left: Plan for SQ-REST-SIZE; Right: Attributed Flow Graph Projection



Constraints:

1. output of "f" feedback to input of "f"
2. node-type of "f" is any function
3. "f" is in a loop

Attribute-Transfer:

1. Gen's Function := whatever type matches "f"
2. Generation's ce := outside-ce of the innermost loop containing the ce of "f"

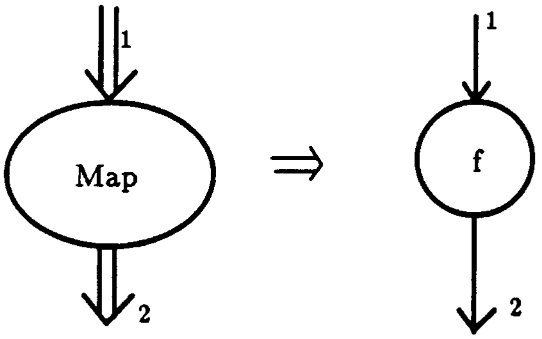
Figure 2.45: Rule for Generation Plan

a *Count* is a generator which takes an integer and the function “1+”, and outputs a sequence of consecutive integers beginning with the input integer.

The grammar rule for Generation is shown in Figure 2.45. (Data flow edges which are viewed as carrying a sequence are drawn with a double-lined arrow.) The feedback constraint specifies that the output of CDR feeds back to the input. Thus, the Generator repeatedly applies the generating function to the previous output of that function.

There are several points worth noting about the rule for Generation that are relevant to the other rules for basic loop plans as well. First, even though the plans all take a function as an input, when each plan is converted to a grammar rule, the function is not represented explicitly in the structure of the rule. Rather, it is handled in attributes and constraints on attributes. The rule has a constraint on the type of function that may be used as an input into the plan. In the case of Generation, the function can be any operation in Common Lisp that is in a loop and that satisfies the feedback constraints. However, it will be seen for other plans, such as Filters, that the function will be constrained to be a predicate. The reason for handling the functions this way and the implications of this will be discussed after all the rules for loop plans have been described.

Because the function used by the Generation plan may be any operation in Common Lisp, rather than having a grammar rule for each type of operation, the Recognizer makes use of generalized node types. In order for a higher level rule which uses a Generation nonterminal to be able to determine which function was used by the Generator, one of the attribute-transfer specifications is the transfer of the function type to the left-hand side node. Thus, when Cdr



Constraints:

1. node-type of "f" is any function
2. "f" is in a loop

Attribute-Transfer:

1. Map's Function := whatever type matches "f"
2. Map's ce := outside-ce of the innermost loop containing ce of "f"

Figure 2.46: Rule for Map

Generation is found in POS-ELEMENTS, it will be reduced to a Generation non-terminal node with an attribute specifying that CDR is the generating function.

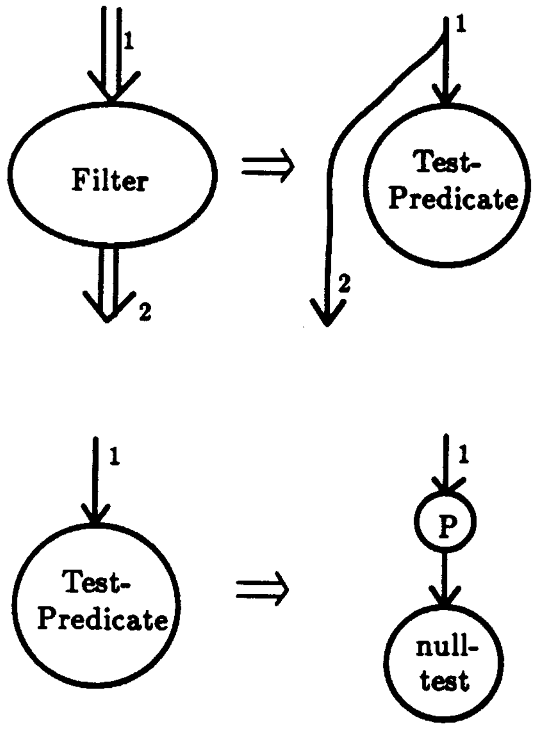
Another point to note about the rule for Generation (which is true of all the rules for the basic loop plans) is that in the attribute-transfer, the left-hand side node's control environment becomes the outside-ce of the innermost loop in which **f** is found in. This is because when a loop is temporally abstracted, it becomes straight-line and is placed in the control environment of the surrounding code.

Map

The plan for *Map* takes a temporal sequence and a function as inputs and gives a sequence as output. The output sequence is the series of values produced by applying the function to each element in the input sequence. The grammar rule for Map is shown in Figure 2.46.

The only constraint that the rule for Map enforces is that **f** be in a loop. With such a loose constraint, almost any operation in a loop can be seen as a Map, including CDR in POS-ELEMENTS (where it is also identified as a Generation), since they all use a sequence of values and output a sequence of values. It is a feature of the Recognizer that parts of the program can be seen as playing more than one role.

As has already been pointed out, the function POS-ELEMENTS contains a Map which takes the function CAR and the sequence of lists generated by the Cdr-Generation and outputs the sequence consisting of the CAR of each sublist.



Constraints:

1. Whatever uses output is in a ce $\sqsubseteq$ success-ce of "Test-Predicate"
2. "Test-Predicate" is in a loop

Attribute-Transfer:

1. predicate := predicate of "Test-Predicate"
2. Control-env. := outside-ce of innermost loop containing "Test-Predicate"

Constraints:

1. "P" is any primitive predicate

Attribute-Transfer:

1. Predicate := whatever matches with "P"
2. Success-ce := failure-ce of "null-test"
3. Failure-ce := success-ce of "null-test"
4. Control-env. := ce of "null-test"

Figure 2.47: Rule for Filter

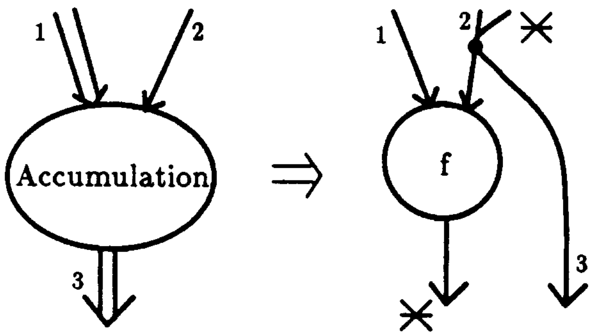
Filter

The *Filter* plan takes a temporal sequence and a predicate and outputs a restricted subset of the input sequence. The output sequence consists of all terms in the input sequence for which the predicate is true. For example, in POS-ELEMENTS, there is a Filter whose predicate is PLUSP and whose input sequence is the sequence given by the Car-Mapping. The output sequence is the series of values of "E" given to CONS in the following statement:

```
(SETQ NEW-L (CONS E NEW-L))
```

The grammar rule for Filter is shown in Figure 2.47. This rule makes use of a new type of constraint on an edge, which requires that the data carried by the edge must only be used by an operation which is in a particular control environment.

The rule for "Test-Predicate" is also given in Figure 2.47. By using this nonterminal, the rule for Filter specifies that the filtering predicate may be any primitive test predicate. (Note that the rule for "Test-Predicate" makes use of generalized node types.) As will be seen, this nonterminal is used in many of the rules for other loop plans besides Filter.



Constraints:

1. output of "f" feedsback to second input of "f"
2. node-type of "f" is any binary
   aggregative function (e.g., cons, plus, times)

Attribute-Transfer:

1. Accum's Function := whatever type
   matches "f"
2. Ce := outside-ce of
   the innermost loop containing "f"

Figure 2.48: Rule for Accumulation

Accumulation

The *Accumulation* plan takes a binary aggregative function (such as CONS, +, or *), a temporal sequence, and an initial value. On each repetition of a loop containing an Accumulation, the function is applied to the current element of the input sequence and the result of the function on the previous iteration (or the initial value, if on the first iteration). Thus, the output is the sequence of values, starting with the initial value, resulting from repeatedly applying the function to the current element in the input sequence and the result of the previous application. For example, the function POS-ELEMENTS contains an Accumulation, called *List-Accumulation* whose function is CONS. It receives the stream of values from the Filter and the initial value NIL. The output sequence is the series of values NEW-L takes on as input to CONS in the statement below, over all iterations of the loop.

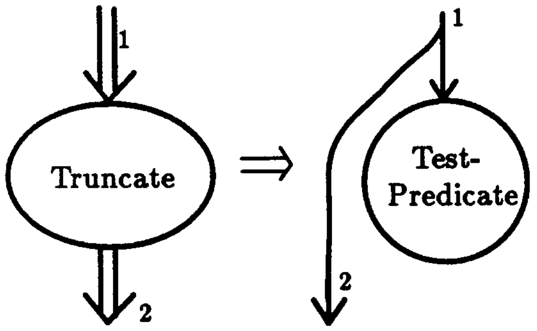
```
(SETQ NEW-L (CONS E NEW-L))
```

Iterative-Aggregation is another type of Accumulation. In Iterative-Aggregation, the accumulation operation is an aggregative function which is commutative, associative, and has identity elements and the initial value is the identity element for that function (e.g., sum, product, union, or intersection).

The rule for Accumulation is shown in Figure 2.48. The feedback constraint indicates that the Accumulation function is repeatedly applied to the current value of the input sequence and the output of the previous application of that function.

Truncate

The *Truncate* plan takes a sequence and a predicate. The output sequence is a subsequence of the input sequence consisting of all terms of the input sequence up to, but not including



Constraints:

1. Whatever uses output is in a
   ce $\sqsubseteq$ failure-ce of "Test-Predicate"
2. "Test-Predicate" is an Exit-Predicate
   (i.e., Success-ce $\sqsubseteq$ outside-ce and
   Failure-ce $\sqsupseteq$ feedback-ce)

Attribute-Transfer:

1. Predicate := predicate-type of "Test-Predicate"
2. Termination-ce := success-ce of "Test-Predicate"
3. Continuation-ce := failure-ce of "Test-Predicate"
4. Control-env. := outside-ce of innermost
   loop containing ce of "Test-Predicate"

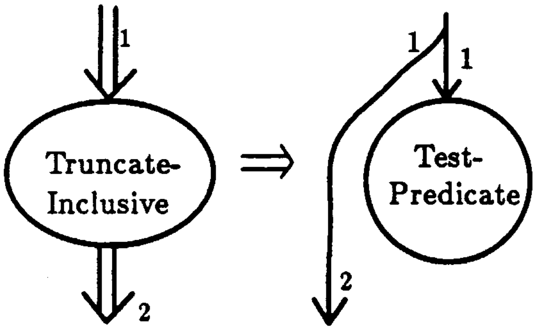
Figure 2.49: Rule for Truncate

the first term that passes the predicate. A typical predicate is the test to see if a list is empty, i.e., NULL.

The grammar rule is shown in Figure 2.49. Two new attributes are used. These are *termination-ce* and *continuation-ce*. The termination-ce is the control environment in which the loop is exited and it corresponds to the success-ce of the exit predicate of Truncate. The continuation-ce is the control environment in which the loop body continues to be executed. It corresponds to the failure-ce of the exit predicate.

Truncate-Inclusive

The *Truncate-Inclusive* plan is similar to Truncate. Its grammar rule is shown in Figure 2.50. The only difference between the plans is that Truncate-Inclusive's output sequence consists of all terms of the input sequence, up to and *including* the first term that passes the predicate. Consequently, the only difference between the rules for Truncate and for Truncate-Inclusive is in the constraint specifying where the output is to be used. Truncate requires that it be used in a control environment less than or equal to the failure-ce of Test-Predicate, while Truncate-Inclusive requires that it be used in a control environment greater than or equal to that of the Test-Predicate.



Constraints:

1. Whatever uses output is in a ce $\sqsupseteq$ control environment of "Test-Predicate"
2. "Test-Predicate" is an Exit-Predicate

Attribute-Transfer:
(same as for Truncate)

Figure 2.50: Rule for Truncate-Inclusive

To see the relationship between Truncate and Truncate-Inclusive, consider the following program.

```
(DEFUN FAKE-PGM (X)
  (A X)
  (LOOP DO
    (C X)
    (COND ((T1 X) (RETURN)))
    (SETQ X (D X)))
  (E X))
```

All values tested by the exit test T1, including the value that passed the test are used by the operation C. This means that the input stream to the Map of C comes from a Truncate-Inclusive in which the truncation predicate is T1. On the other hand, all values tested by T1, except the one that passes the test, are used by D. So, the input stream to the Generation using D comes from a Truncate whose exit predicate is T1.

Co-Truncate

Often two sequences are being used in parallel in a loop and one is truncated based on the occurrence of an element in the other that satisfies the truncating predicate. The *Co-Truncate* plan describes this case. It takes as input two sequences and a predicate. The output sequence is a subsequence of the second input sequence. It consists of all terms of the second input sequence up to, but not including the term that corresponds to the first term of the first input sequence which satisfies the predicate. The grammar rule is shown in Figure 2.51.

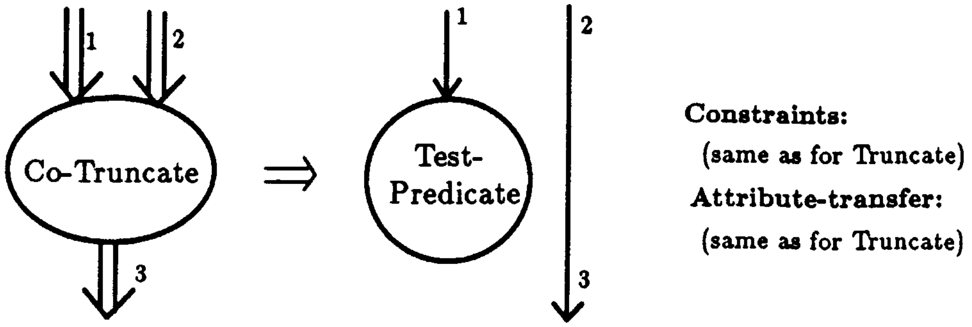


Figure 2.51: Rule for Co-Truncate

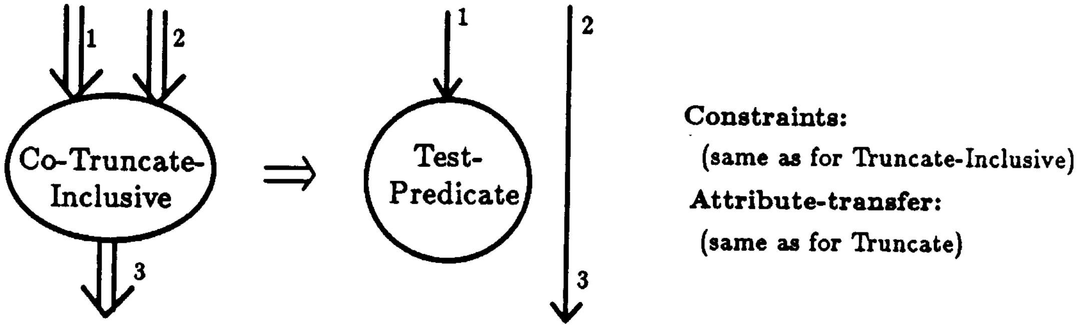


Figure 2.52: Rule for Co-Truncate-Inclusive

Co-Truncate-Inclusive

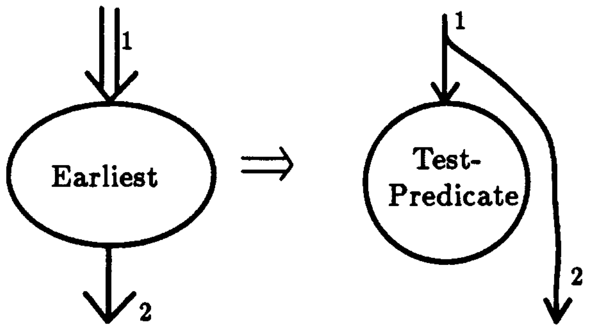
Analogous to Truncate-Inclusive, is Co-Truncate-Inclusive, which has the same structural form as Co-Truncate, but one constraint is different. Instead of the output being used in a control environment $\sqsubseteq$ the failure-ce of the exit predicate, the output must be used in a control environment $\sqsupseteq$ the exit predicate's control environment. This reflects the fact that the output sequence consists of all terms of the second input sequence up to and *including* the term that corresponds to the first term of the first input sequence which satisfies the predicate. The grammar rule for Co-Truncate-Inclusive is given in Figure 2.52.

Earliest

The *Earliest* plan is related to Truncate. It takes as input a sequence and a predicate. Rather than giving a sequence as output, however, the plan returns the first term of the input sequence which passes the predicate. The rule for Earliest is shown in Figure 2.53.

Co-Earliest

Co-Earliest is analogous to Co-Truncate in that it takes two sequences and a predicate

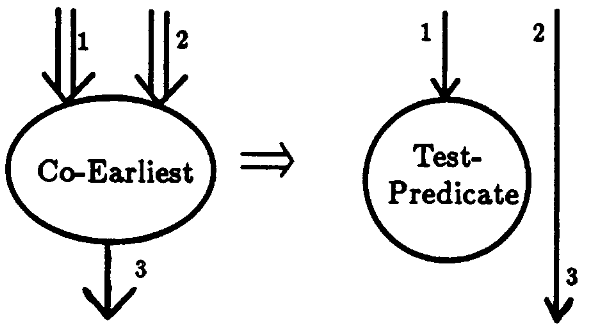


Constraints:

1. Whatever uses output is in a ce $\sqsubseteq$ outside-ce of innermost loop containing ce of "Test-Predicate"
2. "Test-Predicate" is an Exit-Predicate

Attribute-Transfer:
(same as for Truncate)

Figure 2.53: Rule for Earliest



Constraints:
(same as for Earliest)

Attribute-Transfer:
(same as for Truncate)

Figure 2.54: Rule for Co-Earliest

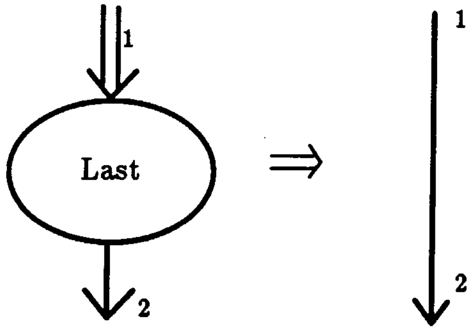
and gives as output the element of the second sequence which corresponds to the first element of the first sequence satisfying the predicate. The only difference is that its result is a single element rather than a sequence. Figure 2.54 displays the rule for Co-Earliest.

Last

Finally, the plan for *Last* takes a sequence and returns the last element of that sequence. It is given by the rule shown in Figure 2.55 and consists almost entirely of a single constraint on how the output is used.

Implications of Dealing with Functional Inputs as Attributes

All loop plans take a function as one of their inputs. For example, Filter takes a predicate to apply to each element of its input sequence. When the plans are converted to grammar rules, the input function becomes a constraint. For example, the input function to Filter is constrained to be some primitive predicate in Common Lisp. The reason that the input function is not represented explicitly as an input to the node in each rule's right-hand side



Constraints:

1. Output is used in a ce $\sqsubseteq$ outside-ce of loop in which input stream is being generated

Attribute-Transfer:

1. Ce := outside-ce ofloop in which input stream is being generated

Figure 2.55: Rule for Last

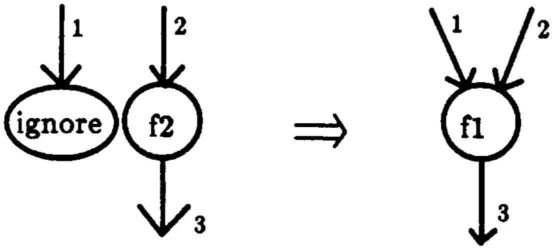
is that the present system cannot analyze programs which take functions as arguments. The Recognizer does not understand, for example, that (CAR X) and (FUNCALL 'CAR X) are the same.

The consequence of this is that the Recognizer must take a somewhat awkward approach to deal with a program in which a loop plan is given an n-ary function as its input function. For example, consider the following function which contains a Map of the operation TIMES.

```
(DEFUN SUM-SCALE (L K)
  (LET ((SCALED-SUM 0))
    (LOOP DO
      (COND ((NULL L) (RETURN)))
      (SETQ SCALED-SUM (TIMES K (CAR L)))
      (SETQ L (CDR L)))))
```

The sequence of values given by the Map of CAR is given to the Map of TIMES, along with the value of K. The analysis of this loop should contain the observation that the operation of multiplying by K is being mapped over the elements of L. To do this the Recognizer must realize that K acts as a constant with respect to the loop. Its value doesn't change over the iterations and K is therefore called a *loop constant.* The operation being mapped over the elements of L isn't simply TIMES, but the function TIMES partially evaluated, having been given one of its arguments, i.e., (LAMBDA (X) (TIMES K X)).

Transformation rules (such as the one shown in Figure 2.56) are used to view a function which is inside a loop and which receives data from a loop constant as a partially evaluated function. The rule's right-hand side specifies the function to be transformed. In Figure 2.56, this is any function in a loop given a loop constant as a first argument. The rule's left-hand



Constraints:

1. "f1" is in a loop
2. Input 1 is a Loop Constant

Attribute-Transfer:

1. "f2" := type of node matching "f1"
2. "f2" is closed-wrt its first argument which receives data flow from [input 1]

Figure 2.56: Rule for Dealing with a Loop Constant

side allows the input from the loop constant to be ignored and gives the transformed node the following attributes during attribute-transfer: *function-type* and *closed-wrt* (which stands for "closed with respect to"). The value of the function-type attribute is the operation type. Closed-wrt associates the number of each of the function's inputs which is receiving data from a loop constant with the port in the graph parsed from which the loop constant is coming. The constraint *partially-evaluated*, when applied to a node, holds true if the node has any values for the closed-wrt attribute. For instance, when the right-hand side of the rule in Figure 2.56 is recognized and replaced by the rule's left-hand side, the node **f2** is seen as partially evaluated. The function which matched **f1** is closed with respect to its first argument. Its first argument receives data flow from the same port that is connected to input 1 of the left-hand side graph.

The Recognizer must deal with partially evaluated functions by using constraints because it cannot analyze programs with functional arguments. If the Recognizer were to acquire this ability, then the partially evaluated functions may be dealt with in a more straightforward way. In particular, rather than having constraints specify the type of partially evaluated functions and how they are closed, this information may be represented explicitly in *Funcall* and *Close* nodes. "Funcall" nodes correspond to the FUNCALL operation in Common Lisp which gives the result of applying the first argument (which is a function) to the rest of the arguments. "Close" nodes specify that a function is closed with respect to some argument. They give another function as output.

For example, if these nodes were available, the rule for Map would be the one shown in Figure 2.57. The left-hand side of the rule corresponds more closely with the intuitive notion of Map which is that it has *two* inputs: a function and a sequence to map the function over. This requires that the function calls in a loop be seen as being achieved via FUNCALL (i.e.,

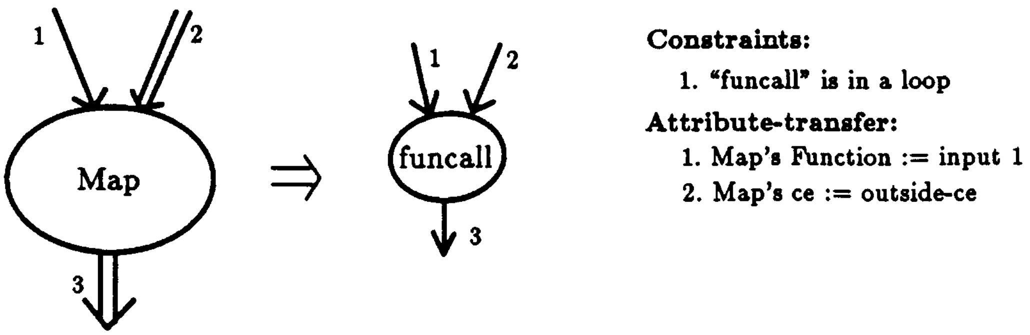

1
2
Map
3
1
2
funcall
3
Constraints:
1. "funcall" is in a loop
Attribute-transfer:
1. Map's Function := input 1
2. Map's ce := outside-ce

Figure 2.57: Rule for Map When Using Funcall and Close Nodes

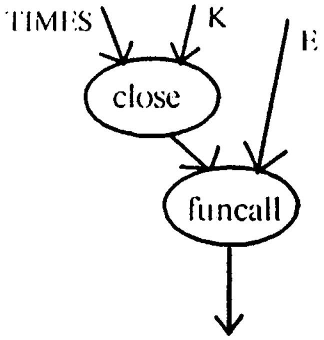
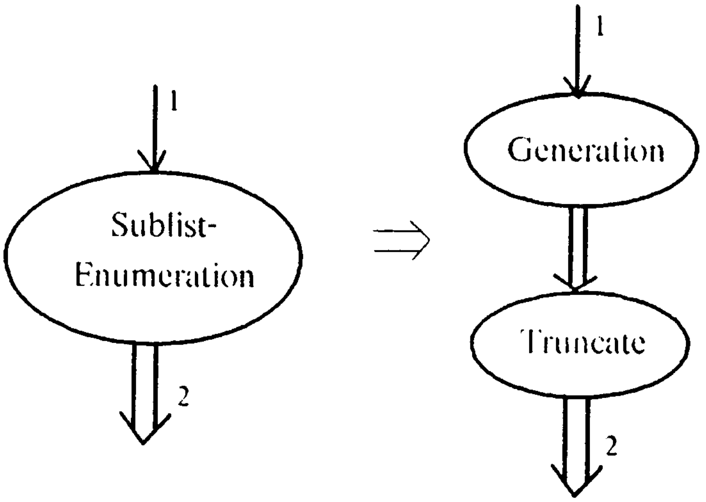


Figure 2.58: Graph in which Map Occurs

(CAR X) must be seen as (FUNCALL 'CAR X), as in Figure 2.57). Additionally, any functions which are applied to loop constants must be seen as being closed with respect to the loop constant. For example, the expression (TIMES K E), where K is a loop constant but E is not, must be viewed as (FUNCALL (CLOSE 'TIMES K) E), as is shown in Figure 2.58. In this way, a Map may be recognized in the graph in Figure 2.58, yielding the analysis that the operation of multiplying by K is being mapped over the sequence of values E takes on over the iterations of the loop.

Because of time constraints and the complexity this technique would add to the Recognizer, this approach has not yet been taken. It is worthwhile pursuing in the future. For now, however, the attribute and transformation mechanisms will be used to deal with the problem of partially evaluated functions.



Constraints:

1. Generation and Truncate co-occur
2. Generation's function is "cdr"
3. Truncate's predicate is "null"

Attribute-Transfer:

1. Ce := ce of either of the nodes

Figure 2.59: The Rule for Sublist-Enumeration

Loop Examples

This section gives examples of higher-level plans which contain the basic loop plans just described.

Sublist-Enumeration

A common cliché involving the loop plans Generation and Truncate in which the input functions are CDR and NULL, respectively, is the *Sublist-Enumeration* cliché. The grammar rule for this cliché is shown in Figure 2.59. It is found in the following program whose flow graph is shown in Figure 2.60.

```
(DEFUN SAMPLE-SLE (L)
  (LET ((E NIL))
    (LOOP DO
      (COND ((NULL L) (RETURN)))
      (SETQ L (CDR L)))))
```

In Sublist-Enumeration, the successive sublists of the input are generated by the Cdr-Generation. The Truncate applies the predicate NULL to each of the sublists generated by the Cdr-Generation. As long as the predicate is not passed (i.e., the sublist tested is not NIL), Truncate will give as output the sublist it tested. Thus, this pattern of loop plans implements

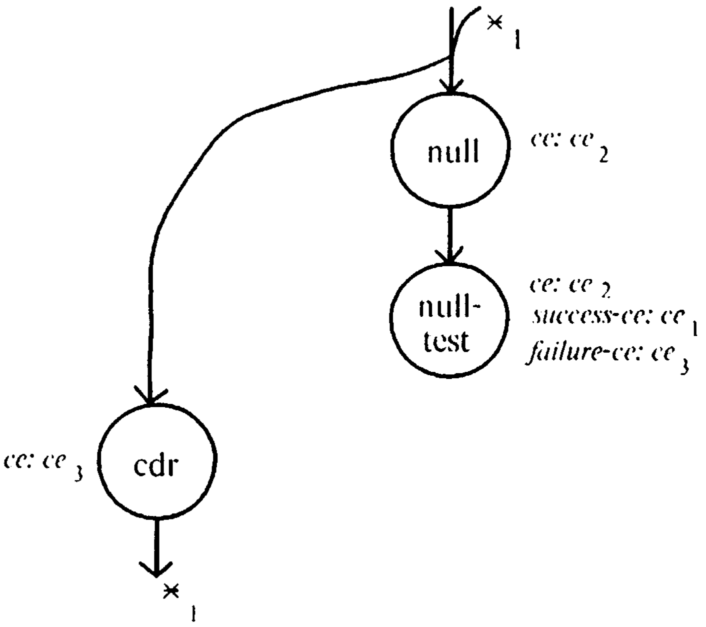


Figure 2.60: The Flow Graph for SAMPLE-SLE

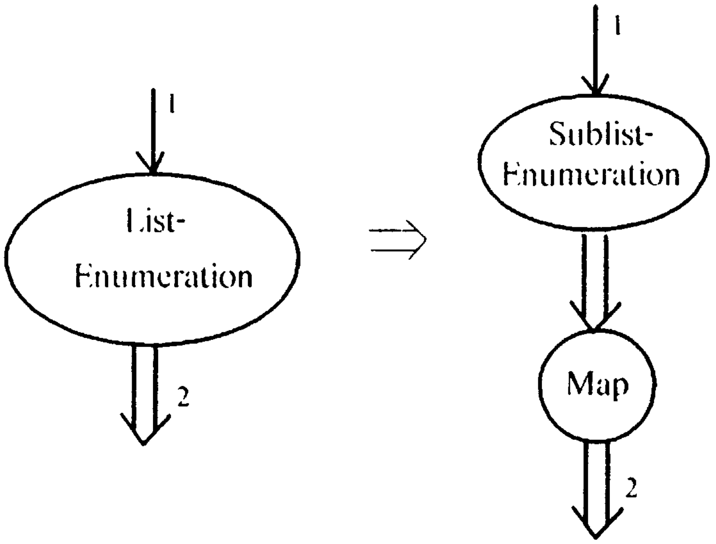
the operation of "cdr-ing down" a list, making each sublist available to operations in the loop body, until the end of the list is reached.

List-Enumeration

Another common cliché , called *List Enumeration*, makes use of the Sublist-Enumeration cliché along with a Map whose input function is CAR. The grammar rule for this cliché is shown in Figure 2.61. It is found in the following program whose flow graph is shown in Figure 2.62.

```
(DEFUN SAMPLE-LE (L)
  (LET ((E NIL))
    (LOOP DO
      (COND ((NULL L) (RETURN)))
      (SETQ E (CAR L))
      (SETQ L (CDR L)))))
```

In List-Enumeration, the successive sublists of the input list are enumerated by Sublist-Enumeration. The first element of each sublist is given in the output sequence of the Map



Constraints:

1. Sublist-Enumeration and Map co-occur
2. Map-function is "car"

Attribute-Transfer:

1. Ce := ce of either of the nodes

Figure 2.61: The Rule for List-Enumeration

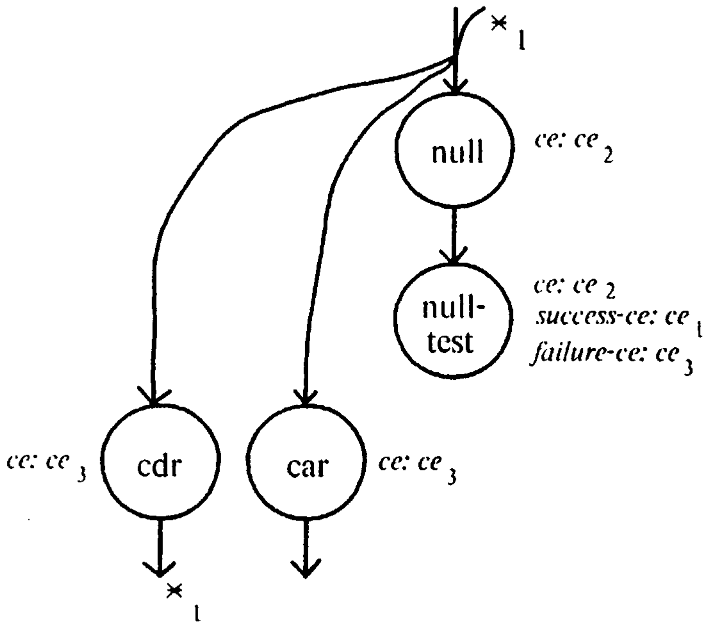


Figure 2.62: The Flow Graph for SAMPLE-LE

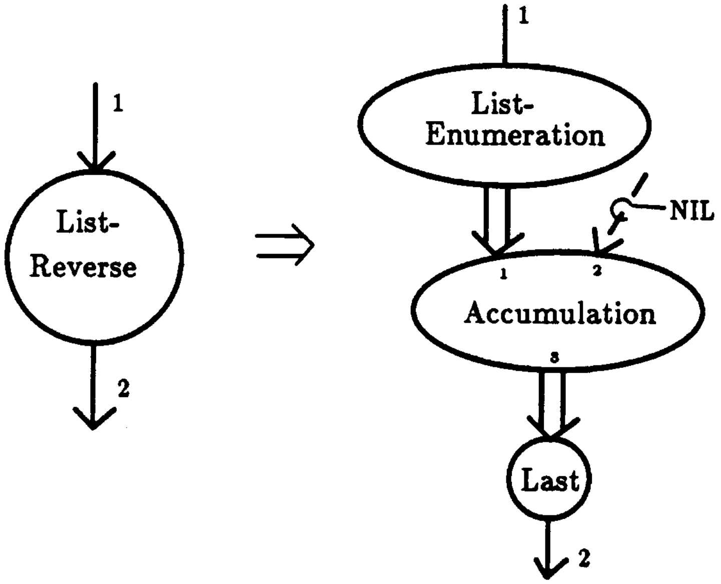
which maps the function CAR over all the sublists given by the Sublist-Enumeration. Thus, List-Enumeration "cdr's down" a list, making each element available to operations in the loop body.

As another example, consider the following code in which a List-Enumeration of SL is recognized, but not a List-Enumeration of L. This is because, although the structural form of a List-Enumeration of L is contained in the program, the co-occurrence constraint between the Sublist-Enumeration and Map has been violated. It is right that the Recognizer recognizes only a List-Enumeration of SL, because although the program "cdr's down" L (via Sublist-Enumeration), it doesn't make each element of L available (using CAR) within the loop.

```
(DEFUN NESTED (SL)
  (LET ((L NIL)
        (E1 NIL))
    (LOOP DO
      (COND ((NULL SL) (RETURN)))
      (SETQ L (CAR SL))
      (SETQ E1 (CAR L))
      (LOOP DO
        (COND ((NULL L) (RETURN)))
        (SETQ L (CDR L)))
      (SETQ SL (CDR SL)))))
```

List-Reverse

The plan for reversing a list uses List-Enumeration, Accumulation, and Last, as is shown in Figure 2.63. Each element enumerated is accumulated using the function CONS. The initial value of the Accumulation is Nil. The final element of the sequence generated by the Accumulation is made available outside the loop by Last. Since each element is "consed" onto the front of the list resulting from the previous application of CONS, the elements are added to the accumulated list in reverse order.



Constraints:

1. List-Enum, Accum, and Last co-occur
2. Accum's Function is "cons"
3. Accum's initial value (input port 2) is the constant NIL

Attribute-Transfer:

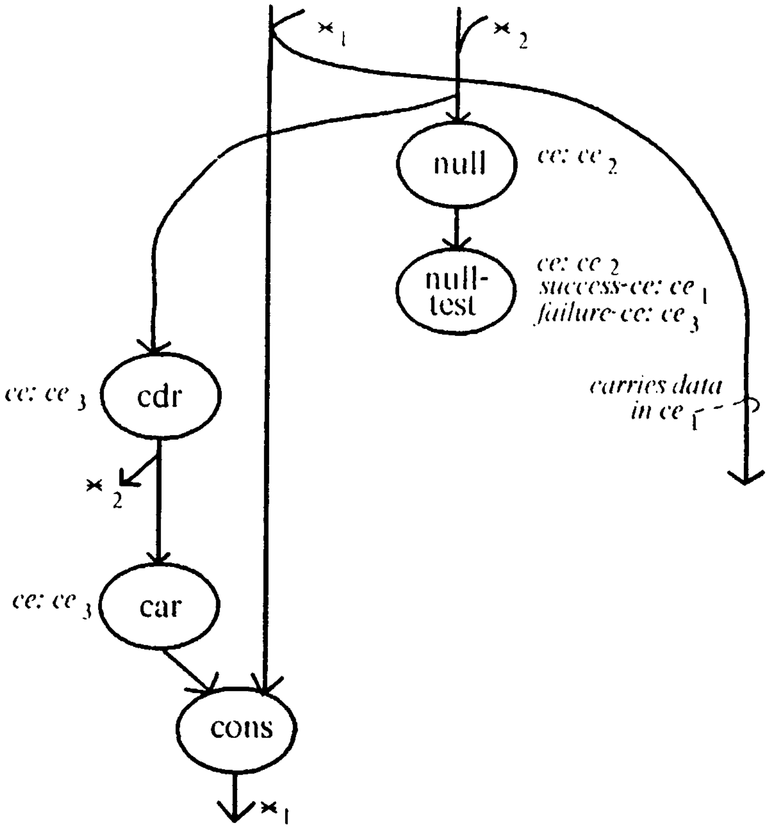
1. Ce := ce of any of the nodes

Figure 2.63: The Rule for List Reverse

An example of a program in which List Reverse is recognized is:

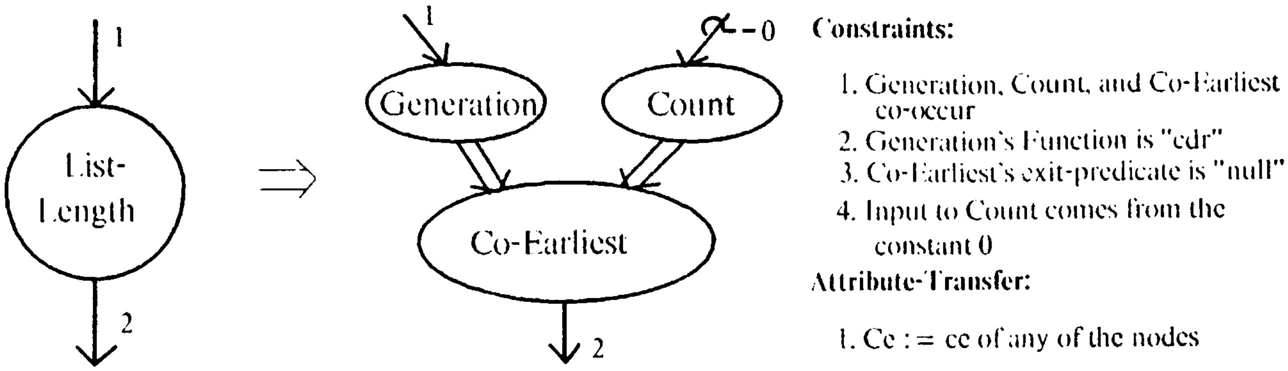
```
(DEFUN REV (L)
  (LET ((NEW-L NIL))
    (LOOP DO
       (COND ((NULL L) (RETURN NEW-L)))
       (SETQ NEW-L (CONS (CAR L) NEW-L))
       (SETQ L (CDR L)))))
```

If the last two statements of the program were switched as in the code and flow graph in Figure 2.64, the List-Reverse cliché wouldn't and shouldn't be recognized because there is a structural mismatch. The first element of the input list is not included in the output list and the value of L which passed the exit test will be included in the input sequence to the Accumulation. Thus, the output will be a list starting with NIL, followed by the reverse of the CDR of the input list. The reason is that the List-Enumeration generates the next element of its output sequence while the current value is still needed. This is a common bug. A library of bugs may be built by collecting structural variations of plans such as this one.



```
(DEFUN BUG-REV (L)
  (LET ((NEW-L NIL))
    (LOOP DO
      (COND ((NULL L) (RETURN NEW-L)))
      (SETQ L (CDR L))
      (SETQ NEW-L (CONS (CAR L) NEW-L)))))
```

Figure 2.64: Buggy Code for Reversing a List and Its Flow Graph



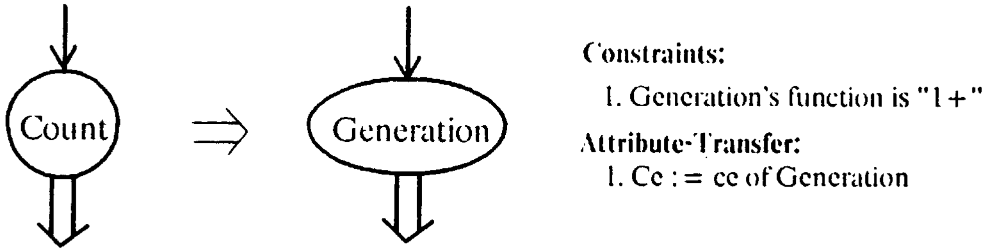
Constraints:

1. Generation, Count, and Co-Earliest co-occur
2. Generation's Function is "cdr"
3. Co-Earliest's exit-predicate is "null"
4. Input to Count comes from the constant 0

Attribute-Transfer:

1. Cc := cc of any of the nodes

Figure 2.65: The Rule for List-Length



Constraints:

1. Generation's function is "1+"

Attribute-Transfer:

1. Cc := cc of Generation

Figure 2.66: The Rule for Count

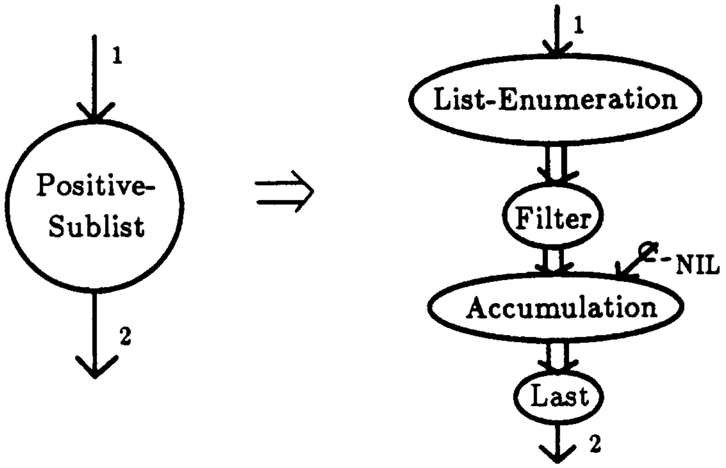
List-Length

The rule for *List-Length*, shown in Figure 2.65, uses Co-Earliest and Count. Count's rule is shown in Figure 2.66. It generates a sequence of numbers, starting with the input number.

In List-Length, two sequences are generated in parallel: the sequence of sublists of L generated by a Cdr-Generation and the sequence of natural numbers. The output is the natural number corresponding to the first sublist (NIL) which passes Co-Earliest's predicate (NULL).

Positive-Sublist

The plan for *Positive-Sublist* makes use of the Filter plan. The grammar rule is shown in Figure 2.67. The elements of the input list are generated by the List-Enumeration and filtered so that only the positive elements are given as input to the Accumulation. This plan may be recognized in the function POS-ELEMENTS shown again below.



Constraints:

1. List-Enumeration, Filter, and Accumulation co-occur
2. Accumulation's Function is "cons"
3. Filter's Predicate is "plusp"
4. Accumulation's initial value is the constant NIL

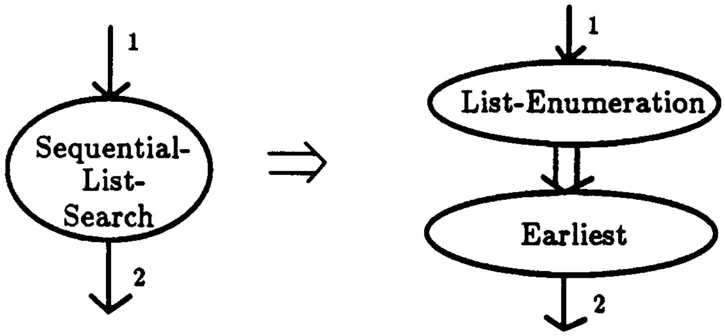
Attribute-Transfer:

1. Ce := ce of any of the nodes

Figure 2.67: The Rule for Positive-Sublist

```
(DEFUN POS-ELEMENTS (L)
  (LET ((NEW-L NIL)
        (E NIL))
    (LOOP DO
      (COND ((NULL L) (RETURN NEW-L)))
      (SETQ E (CAR L))
      (IF (PLUSP E)
          (SETQ NEW-L (CONS E NEW-L)))
      (SETQ L (CDR L)))))
```

Positive-Sublist will not be recognized in the following buggy version of POS-ELEMENTS (IF has been changed to UNLESS). This is because the constraint of the Filter restricting the control environment in which its output may be used is violated. In particular, the output is used in the failure-ce of the filtering predicate, rather than in its success-ce. The Accumulation function CONS uses the tested element in the failure-ce of the Filter's predicate. This means only non-positive elements are accumulated.



Constraints:

1. List-Enum and Earliest co-occur

Attribute-Transfer:

1. Ce := ce of either node
2. Success-ce := Earliest's termination-ce
3. Failure-ce := List-Enum's termination-ce
4. Predicate := predicate of Earliest

Figure 2.68: The Rule for Sequential-List-Search

```
(DEFUN BUGGY-POS-ELEMENTS (L)
  (LET ((NEW-L NIL)
        (E NIL))
    (LOOP DO
      (COND ((NULL L) (RETURN NEW-L)))
      (SETQ E (CAR L))
      (UNLESS (PLUSP E)
        (SETQ NEW-L (CONS E NEW-L)))
      (SETQ L (CDR L)))))
```

Sequential-List-Search

The plan for *Sequential-List-Search* describes a two-exit loop. One exit is taken if an element of the list is found which satisfies some predicate and the other is taken if the list runs out before any element satisfying the predicate is found. Its grammar rule is shown in Figure 2.68.

An example program containing this cliché, where the search predicate is PRIMEP, is shown below. The elements generated by the List-Enumeration are tested by Earliest. If any satisfies Earliest's predicate (in this case, if any are prime), then the element is given as output.

```
(DEFUN PRIME-SEARCH (L)
  (LET ((E NIL))
    (LOOP DO
      (COND ((NULL L) (RETURN NIL)))
      (SETQ E (CAR L))
      (COND ((PRIMEP E) (RETURN E)))
      (SETQ L (CDR L)))))
```

Information about whether the search succeeded is transferred using attribute-transfer specifications to the abstract Sequential-List-Search operation so that this operation may be used as a predicate. If the loop is exited when Earliest's predicate is satisfied, then the search succeeded. This is specified by having the termination-ce of Earliest transferred as the success-ce of Sequential-List-Search. Similarly, if the loop is exited when the list runs out, then the search failed, so the termination-ce of List-Enumeration becomes the failure-ce of Sequential-List-Search.

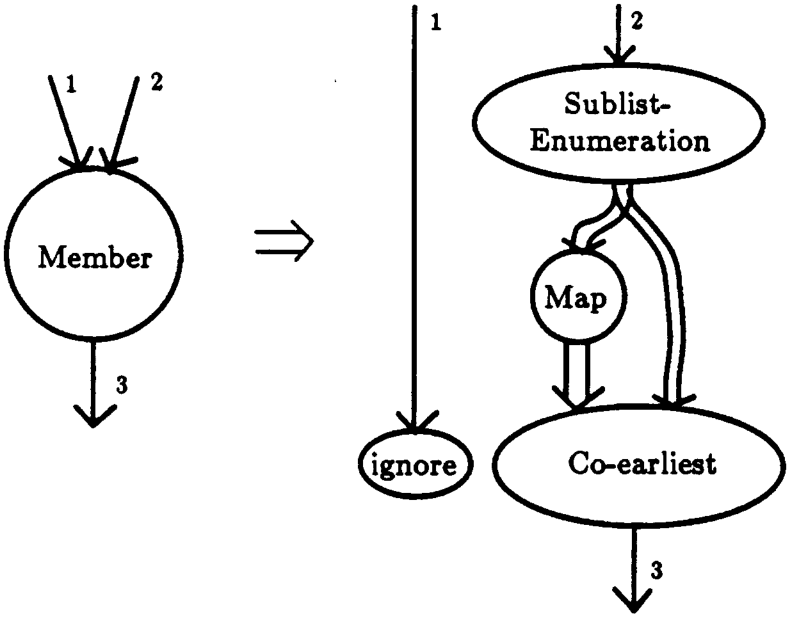
Member

The Common Lisp operation *Member* takes an element and a list and searches for the element in the list using EQ. If the element is found, it returns the sublist of the list beginning with the first occurrence of the element, otherwise, NIL is returned. The rule for the *Member* plan is shown in Figure 2.69.

The plan for Member involves a Co-Earliest loop plan whose predicate is constrained to be the partially evaluated primitive predicate EQ. This predicate must be closed with respect to the input corresponding to the element being searched for (i.e., the first input to the Member plan).

When the predicate is satisfied, the sublist whose CAR satisfied the predicate is returned. This is specified in the grammar rule by requiring that the output of the Sublist-Enumeration be the second input sequence to the Co-Earliest plan.

Since Member is a predicate, there are specifications for how to compute its success and failure control environments, similar to those for Sequential-List-Search.



Constraints:

1. SubList-Enumeration, Map, and Co-Earliest co-occur
2. Map's Function is "car"
3. Co-Earliest's exit-predicate is "eq"
4. "eq" is partially evaluated
5. "eq" is closed-wrt its first arg which receives data flow from input 1

Attribute-Transfer:

1. Ce := ce of any of the nodes
2. Success-ce := termination-ce of Co-Earliest
3. Failure-ce := termination-ce of Sublist-Enumeration

Figure 2.69: Rule for Member

The following program contains the Member plan.

```
(DEFUN MY-MEMBER (ELEM LIST)
  (LOOP DO
    (COND ((NULL LIST) (RETURN NIL))
          ((EQ (CAR LIST) ELEM)
           (RETURN LIST)))
    (SETQ LIST (CDR LIST))))
```

The flow graph projection for MY-MEMBER's plan is shown in Figure 2.70a. In order for the Recognizer to recognize Member in it, the flow graph must be transformed to the flow graph shown in Figure 2.70b.

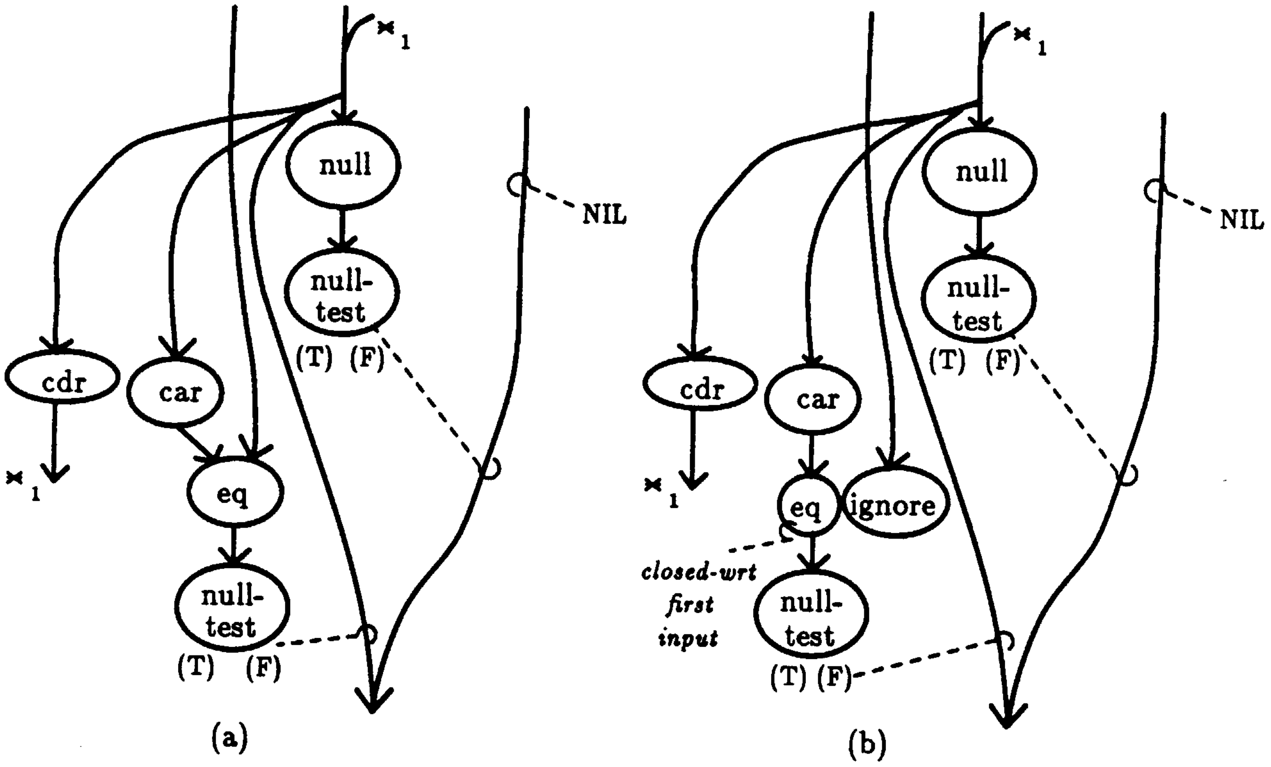


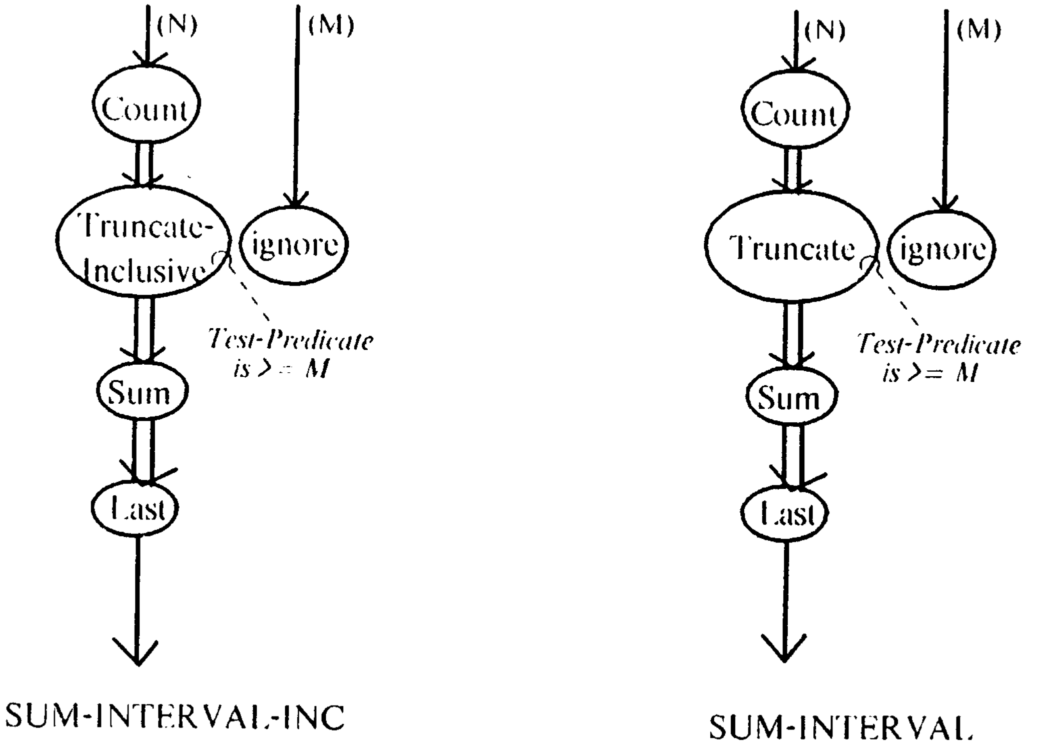
Figure 2.70: Flow Graph (a) and Transformed Flow Graph (b) for MY-MEMBER

An Example Involving Truncate and Truncate-Inclusive

To understand the relationship between the plans Truncate and Truncate-Inclusive, consider the following two programs. The first computes the sum of the integers from N to M, inclusive, while the second computes the sum of the integers from N up to, but not including, M. The temporal abstraction of SUM-INTERVAL-INC and SUM-INTERVAL into compositions of loop plans is shown in Figure 2.71. They differ only in that SUM-INTERVAL-INC uses Truncate-Inclusive, which means M is included in the sum, while SUM-INTERVAL uses Truncate and so does not include M in the sum.

```
(DEFUN SUM-INTERVAL-INC (N M)
  (LET ((SUM 0))
    (LOOP DO
      (SETQ SUM (+ N SUM))
      (COND ((>= N M) (RETURN SUM)))
      (SETQ N (1+ N)))))

(DEFUN SUM-INTERVAL (N M)
  (LET ((SUM 0))
    (LOOP DO
      (COND ((>= N M) (RETURN SUM)))
      (SETQ SUM (+ N SUM))
      (SETQ N (1+ N)))))
```



SUM-INTERVAL-INC

SUM-INTERVAL

Figure 2.71: Temporal Abstraction of SUM-INTERVAL-INC and SUM-INTERVAL

2.3.4 Partial Recognition

It has been shown how the mechanisms of the Recognizer and the extensions made to the parser have made possible the recognition of clichés in programs by parsing. However, the Recognizer cannot always perform program recognition by reducing a program's graph to a single start node. Recognition will in general be partial since programs are rarely made up entirely of familiar forms and data structures. There are usually some unrecognizable parts. Furthermore, the clichés that are recognizable are not always those on the top level of the Plan Library. The Recognizer must search for clichés on *all* levels. Therefore, the grammar used by the Recognizer's parser cannot always have just one start node type which is induced by a single top-level cliché. The start types induced by the clichés would all have differing input and output arities corresponding to the number of inputs and outputs the clichés have.

Recognizing clichés on any level of the library amid unrecognizable sections of a program is done using four techniques:

1. Allowing any nonterminal in the grammar induced by the Plan Library to be treated as a start node type of the grammar. (The grammar may have more than one start node type and the types may have differing arities.)

2. Gathering information from *all* successful parses and subparses, even if the top-level parse that generated them fails.

3. Ignoring unfamiliar parts of a program's graph and parsing the remaining subgraph.

4. Varying the strictness of the constraint language used in the grammar.

The first two techniques are made possible by minor, yet powerful, modifications made to the parser. Due to the first, the input to the parser is now the graph to parsed, the grammar, and a list of start types for that grammar. The second technique is made possible by incorporating dynamic programming into the parser. Not only successful parses, but all parses ever started are recorded along with their outcomes. This technique serves a two-fold purpose. First, because successful subparses are recorded, the Recognizer doesn't require a complete parse in order to recognize something. Second, dynamic programming prevents the same parse from being attempted more than once. Thus the Recognizer may work more efficiently.

The last two techniques listed above (numbers 3 and 4) will be described in detail in the next two sections.

Partial Parses

In order to find familiar features in the midst of unrecognizable parts of the program, the unfamiliar parts of the program's graph must be ignored and the remaining subgraph parsed. Since the unrecognizable parts cannot be identified before parsing is done, the Recognizer systematically ignores all parts of the program's graph in turn. It does this by computing all possible initial positions of the parser's read head and starting a parser at each. Because the parser's read head scans a graph from left to right, any section of the graph which is to the left of the initial head position will be ignored by the parser.

The computation of all possible initial head positions is performed in two stages. First, all *leading edge cutsets* are computed. These are sets of edges which cut through a graph, separating it into two pieces. In connected flow graphs, these correspond to cutsets in the conventional sense that they are a minimal set of edges which separate a graph into two connected components. Leading edge cutsets are therefore generalizations of cutsets to the type of flow graphs used by the Recognizer. Formally, a leading edge cutset is a collection of edges which have the following restrictions:

- all edges in the set point in the same direction
- any path from the input edges of the original graph to its output edges must contain exactly one of the edges in the leading edge cutset
- the edge set is minimal in that no proper subset of the edge set obeys these restrictions.

For example, the graphs in Figure 2.72 show all possible leading edge cutsets through a graph. (The method for computing these will be explained shortly.) The reason they are called "leading edge" cutsets is that they are each to be treated by the Recognizer as input edges of a subgraph. Everything to the left of them is ignored by starting parses at each cutset. This is done in the second phase of computing initial head positions.

To start parsers on each leading edge cutset, all permutations of the edges of the cutset must be produced for each start node type. Each permutation is used as the initial read head

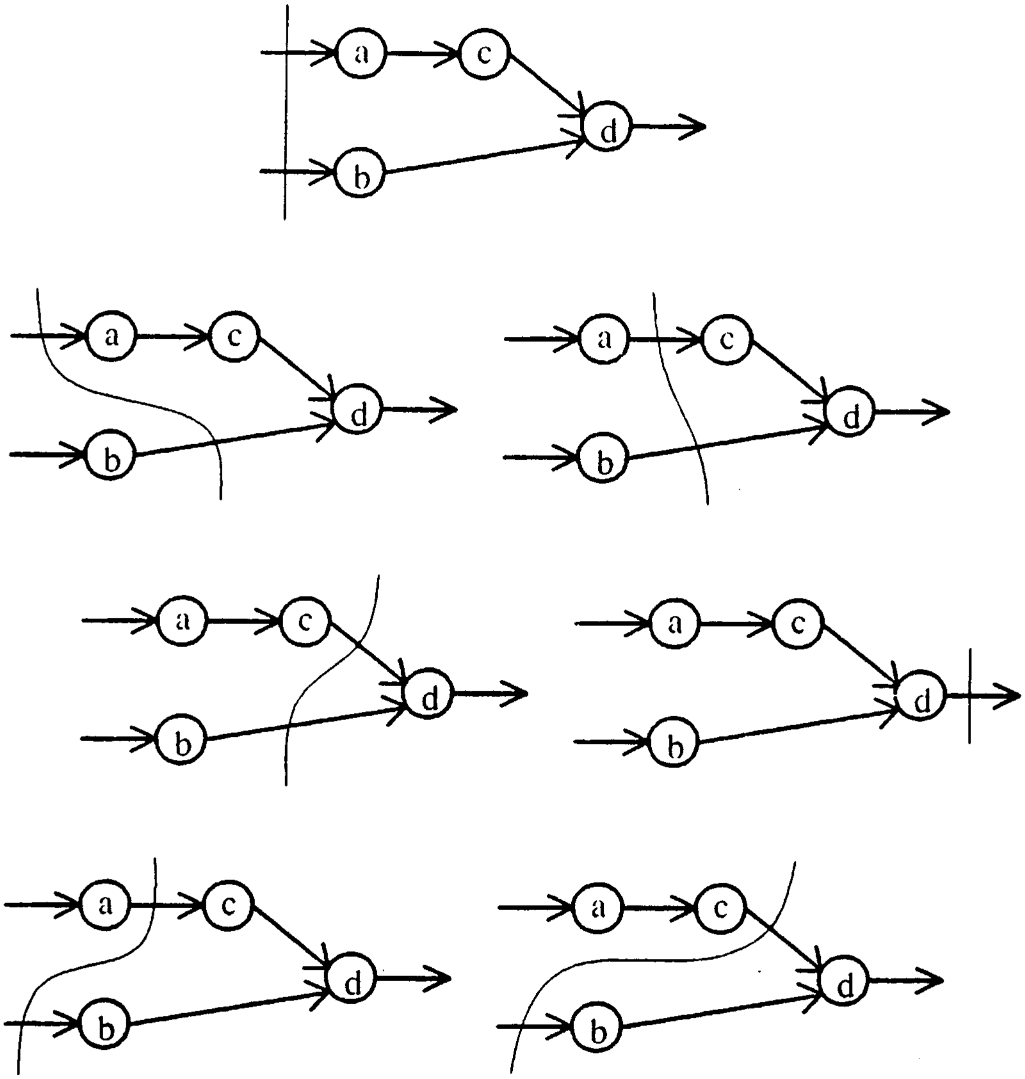


Figure 2.72: All Possible Leading Edge Cutsets

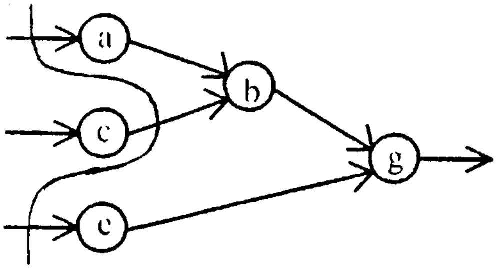


Figure 2.73: A Possible Read Head Position in a Flow Graph

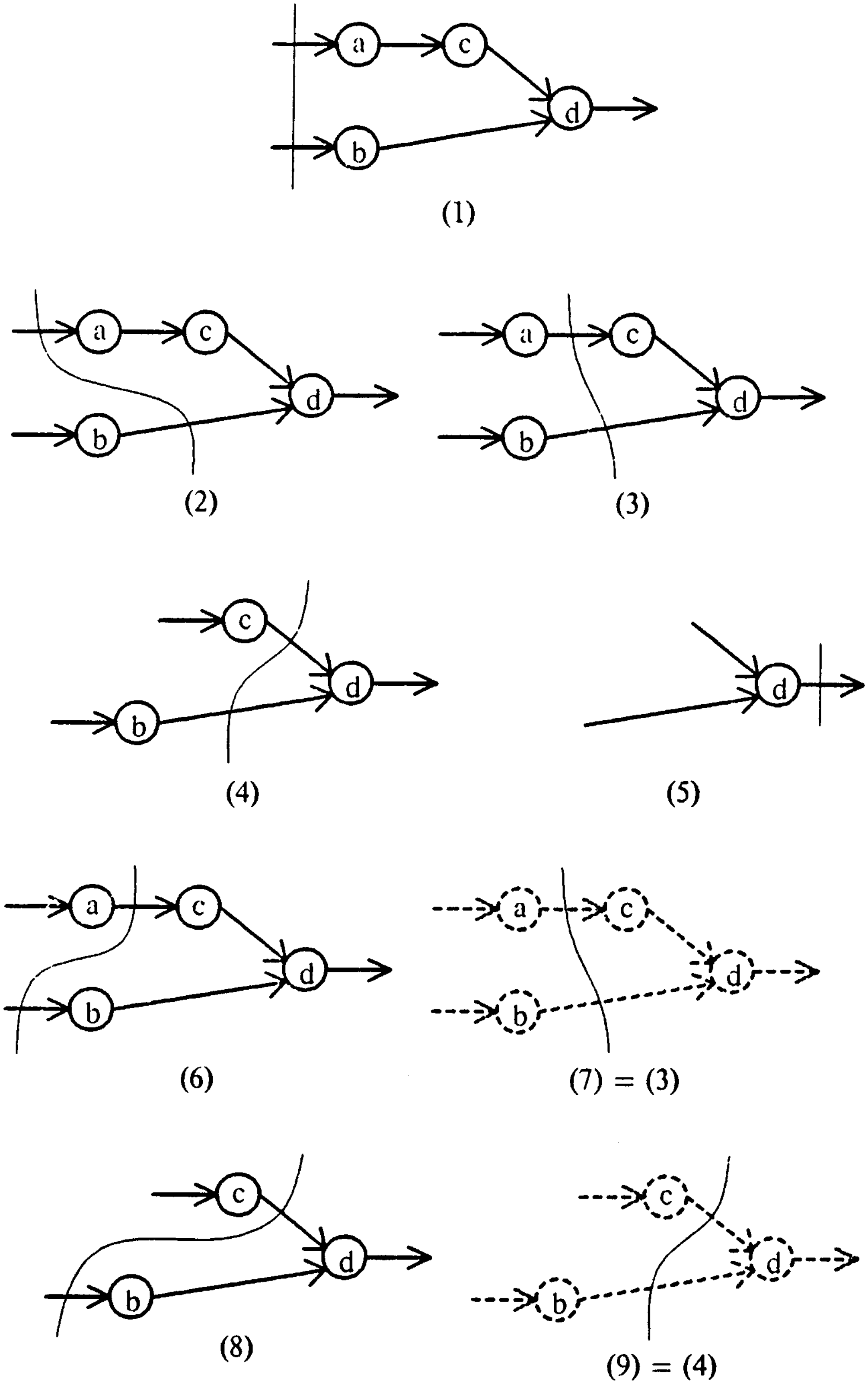
position of a parser. That is, it is the set of edges which are "seen" by the parser as the leftmost edges of a graph to be parsed. Each permutation must be of size equal to the number of inputs to the corresponding start node. This is because the start node type must have the same arity as the input graph in order to match with it. Therefore, if there are n edges in the leading edge cutset and the input arity of the start type is k, then all k-permutations of the n edges on the leading edge cutset are computed.

Since the grammar may have many start types, these combinations of edges in the edge-set must be generated for each start type. (For efficiency, the start types are grouped according to input arity so that combinations of any particular size are only generated once.)

Generating Leading Edge Cutsets

The restrictions on leading edge cutsets arise from the way the leading edge cutsets are generated, that is, by using the graph reading mechanism of Brotsky's parser. The parser reads nodes by advancing over any one of the nodes all of whose edges are on the current head position. It never reads a node unless all of its predecessors have been read. A node's predecessors have been read if all of the node's incoming edges are in the current head position. For example, in Figure 2.73, the read head is positioned before the nodes **a**, **b**, and **e**. It may step over either **a** or **e**, but not over **b** because not all of **b**'s predecessors have been read yet. By choosing "any one of" the nodes on the head position, the read head picks an arbitrary path through the graph. In order to find all leading edge cutsets, the Recognizer must take all possible paths that the read head could take. At each step, the edge set of the current read head position is produced as a leading edge cutset.

The leading edge cutset generator starts positioned to the left of a graph's minimal nodes (at the left-fringe of the input graph) as is shown in Figure 2.74(1). The parser's read head



(1)

(2)

(3)

(4)

(5)

(6)

(7) = (3)

(8)

(9) = (4)

Figure 2.74: Steps Involved in Generating All Leading Edge Cutsets

selects one of the minimal nodes to scan. It is allowed to do so if all of the node's incoming edges are in the current read head position. (See Figure 2.74(2).) The generator then recursively generates the leading edge cutsets of the subgraph whose minimal nodes are those to the right of the current head position (as in Figure 2.74(3) — (5)).

The generator next pretends that the node wasn't scanned (i.e., it "unsteps" the read head back across the node just scanned) and scans one of the other steppable minimal nodes. (See Figure 2.74(6).) It does this until all nodes have been tried.

Because the generation of leading edge cutsets is recursive, there may be an overlap in the cutsets made. In order not to generate duplicate cutsets, the cutsets generated are cached so that when an cutset is made that has already been reached by a different path, the generator will not perform the recursive call of itself on that cutset. In Figure 2.74, step 6 goes directly to step 8 because stepping over node **b** gives the same leading edge cutset as in step 3.

An Example

As an example of recognizing a cliché by starting parsers everywhere throughout a program's graph, consider trying to find the Sum-of-Squares cliché in the following code:

```
(DEFUN F3 (X Y)
  (C (+ (SQUARE (A X))
        (SQUARE (B Y)))))
```

The flow graph projection of the plan for the function F3 is given in figure 2.75. The leading edge cutsets generated are also shown in F3's plan. The rule induced by the Sum-of-Squares plan definition is given in Figure 2.76.

In order to recognize Sum-of-Squares, in the plan for F3, either the Sum-of-Squares non-terminal must be given as a start type or it must be used in the right-hand side of some higher level cliché which is also recognized in F3 (enough to see Sum-of-Squares). If it is a start type, Sum-of-Squares will be found when the initial head position is the set of the two edges coming into the nodes of type *square* (the leading edge cutset marked with an asterisk). The parser "sees" the input graph in this case look like the graph in Figure 2.77. The recognizer for Sum-of-Squares will complete and the Recognizer will record the derivation. Sum-of-Squares will be recognized even though the parse will ultimately fail when the read head scans C and no recognizer is active to account for it.

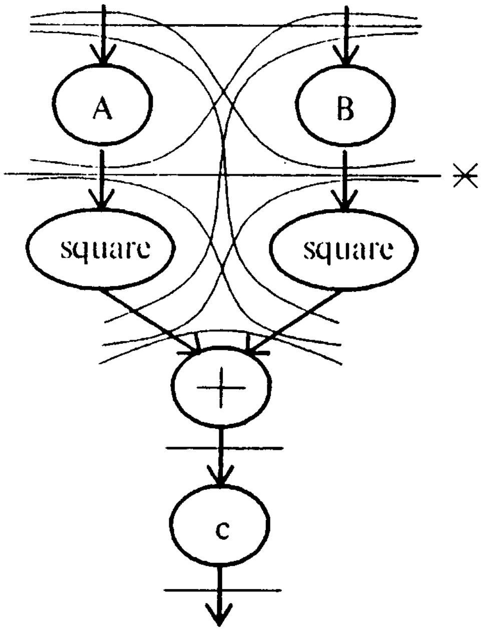


Figure 2.75: F3's Flow Graph Projection and Leading Edge Cutsets

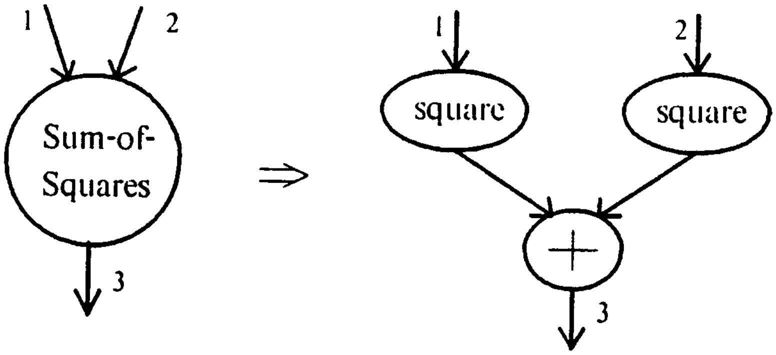


Figure 2.76: The Rule for Sum-of-Squares

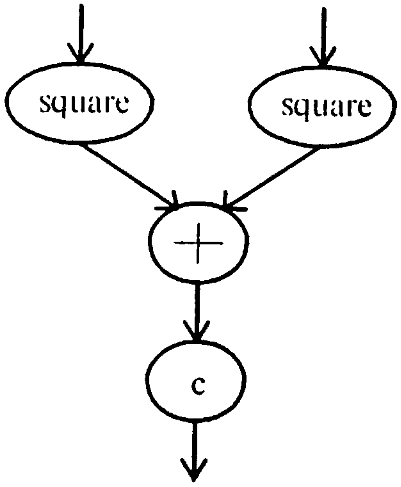


Figure 2.77: Part of the Flow Graph for the Function F3

Conditional Recognition

So far, three techniques have been set forward which facilitate the recognition of clichés even though the program may not be reducible to a single nonterminal node. In particular, the Recognizer performs partial recognition by allowing nonterminals on all levels of the grammar to be start types, by recording all parses and subparses that succeed even if a higher level parse fails, and by starting parses all over the input graph. This section describes how recognition may be achieved in a fourth way — by having loose constraints on the grammar rules.

Often a cliché's computation may be performed in code only under certain conditions. For example, consider the following program.

```
(DEFUN SOMETIMES-F (X Y)
  (B
    (COND ((>= X Y) (A X))
    (T (C Y))))
```

If the rule shown in Figure 2.78a represented a cliché to be found, the cliché would be *conditionally* recognized in the program's flow graph (shown in Figure 2.78b). The condition it is recognized under is that x ≥ y.

As another example of conditional recognition, the program SOMETIMES-LENGTH (shown below) computes the length of the list L iff L has fewer than 100 elements.

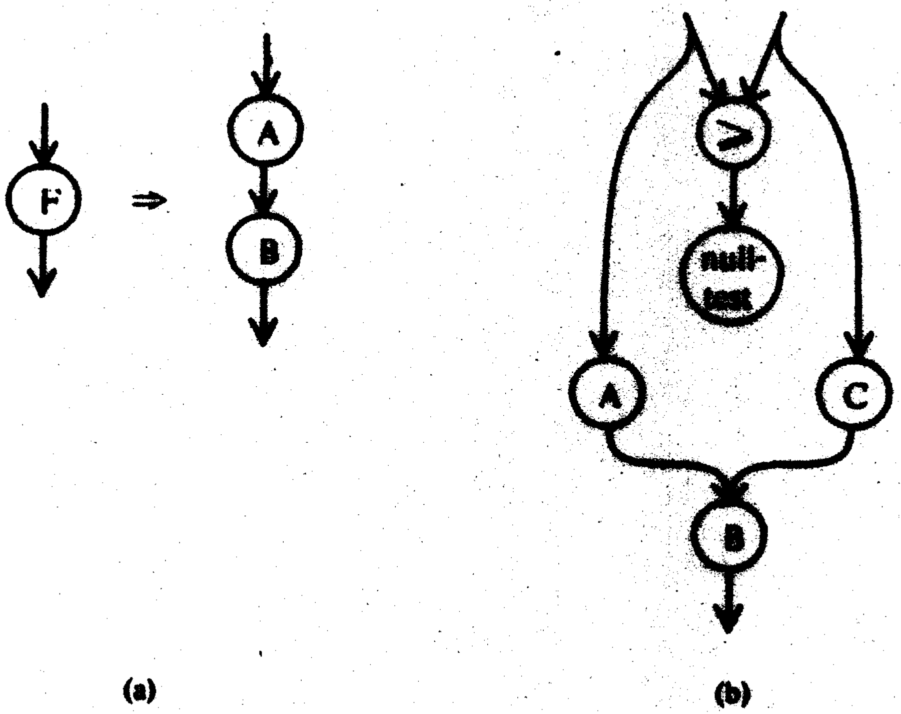


Figure 2.78: Rule for F and Flow Graph for SOMETIMES-F

```
(DEFUN SOMETIMES-LENGTH (L)
  (LET ((I 0))
    (LOOP DO
      (COND ((NULL L) (RETURN I)))
      (SETQ L (CDR L))
      (IF (< I 99)
          (SETQ I (1+ I))
          (RETURN I)))))
```

The strictness of the rules' constraints controls how much can be recognized conditionally. The cliché "F" wouldn't be recognized in SOMETIMES-F if there were a constraint that A and B co-occur. Similarly, List-Length would only be recognized if the loop exit predicate for List-Length had $\sqsubseteq$ in the constraints on its success and failure control environments. If they required that success-ce co-occur with the outside-ce of the loop and failure-ce co-occur with the feedback-ce, then the List-Length cliché would not be recognized.

By varying the strictness of the constraints, the Recognizer can be made to conditionally recognize clichés. The strictness may be varied with how the Recognizer is applied. For example, in a maintenance system, it would be useful to be told that List-Length is computed under certain conditions. However, in a programming tutor, if the program doesn't always perform the required computation, it shouldn't be recognized as doing so. The conditions under which the computation isn't performed are treated as causing bugs. In order to correct the bug, the constraints that are violated must be made to hold. Further research is necessary on the uses of this feature.

2.4 Documentation Generation

Because of the variety of ways clichés may be recognized, the Recognizer uses a heuristic approach to generating documentation. Cyphers [8] shows how explanations can be attached to clichés so that when they occur in a program, automatic documentation can be generated based on the programming knowledge they contain. Each schematized textual explanation fragment contains slots which may be filled in with names of subclichés or variable names used by the clichéd computation. The documentation module which is used to demonstrate the Recognizer's output takes basically this approach. It gives as output an English description of the parse trees generated by the parsing process. For each high level cliché recognized, the system generates a hierarchical, textual description of the code containing the cliché and any subclichés involved. If any data abstraction clues are associated with the rules, they are also given.

When the clichés recognized are disjoint, the documentation lists the clichés and their descriptions. If they are connected by data flow with no unrecognizable sections in between, then they may be seen as being composed. This occurs most frequently in the analysis of loops where a program is temporally abstracted into a composition of loop plans. In this case, the documentation reflects the composition by separating the names of clichés by the word "of" and listing them in the order of how they are composed (i.e., from output up to input). An example is "a Sum of a Filter of a Vector Enumeration" which is the description of a program which has the temporal abstraction shown in Figure 2.79.

Because the parse trees generated in parsing may not always account for the entire program, there are a few heuristics used in producing the documentation. These heuristics determine for example whether the program can be described as *being* an instance of a cliché or as merely *containing* the cliché. In particular, when there are unrecognizable sections of code, i.e., the clichés found only account for part of the program, and the output of the program comes from one of these unrecognizable sections (i.e., it is not the result of some clichéd computation), then the documentation produced says that the program "contains" the cliché. On the other hand, when there are no unrecognizable sections of code or when the results of any unrecognizable sections are only used as input to clichés, the documentation says that the program "is" an instance of the cliché or composition of clichés found. For example, compare

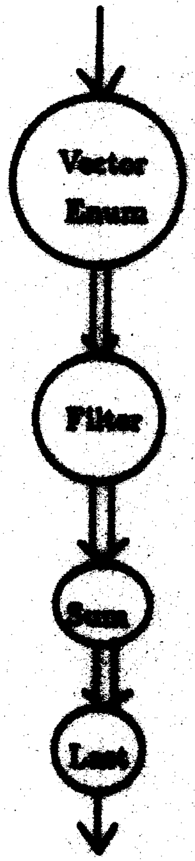


Figure 2.79: A Sum of a Filter of a Vector Enumeration

the two functions and the documentation produced for them in Figure 2.80.[4] The function determines whether the square of the input HYPOTENUSE is approximately equal to the sum of the squares of the two input sides SIDE1 and SIDE2. Because RIGHTP returns the result of the test, it is described as being an instance of an Equality-Within-an-Epsilon, even though there are unrecognizeable sections of the function's code. On the other hand, the function FUNNY-RIGHTP performs the same test on its inputs as RIGHTP does, but it also conditionally returns the ratio of SIDE1 and SIDE2 depending on the result of the test. Because the output of the Equality-Within-an-Epsilon cliché is not given as the result of the function, the function is described as containing the cliché.

A feature of the Documentation Generator that is displayed in the documentation given in Figure 2.80 is that if the output of an unrecognizable subgraph is used as input into a cliché, a Lisp expression is produced which corresponds to the non-clichéd computation and which is used in the schematized text. For example, in the function RIGHTP, the computation "(* 0.02 HYP-SQ)" is not recognized as any cliché and therefore appears in the documentation as a Lisp expression. That is, the comment contains the phrase "Epsilon is (* 0.02 HYP-SQ)".

[4] The function RIGHTP is taken from Problem 3-9 (p. 42) in [47].

```
RIGHTP is an Equality-Within-an-Epsilon of a Square of
HYPOTENUSE and a Sum of Squares of SIDE1 and SIDE2, where
Epsilon is (* 0.02 HYP-SQ).
  The Equality-Within-an-Epsilon determines whether the output
  of the Square and the output of the Sum of Squares differ by
  less than (* 0.02 HYP-SQ).
(DEFUN RIGHTP (HYPOTENUSE SIDE1 SIDE2)
  (LET* ((HYP-SQ (* HYPOTENUSE HYPOTENUSE))
         (DIFF (- HYP-SQ
                  (+ (* SIDE1 SIDE1)
                     (* SIDE2 SIDE2))))
         (DELTA (IF (PLUSP DIFF) DIFF (NEGATE DIFF))))
    (<= DELTA (* 0.02 (* HYP-SQ)))))
```

```
FUNNY-RIGHTP contains an Equality-Within-an-Epsilon of a Square
of HYPOTENUSE and a Sum of Squares of SIDE1 and SIDE2, where
Epsilon is (* 0.02 HYP-SQ).
  The Equality-Within-an-Epsilon determines whether the output
  of the Square and the output of the Sum of Squares differ by
  less than (* 0.02 HYP-SQ).
(DEFUN FUNNY-RIGHTP (HYPOTENUSE SIDE1 SIDE2)
  (LET* ((HYP-SQ (* HYPOTENUSE HYPOTENUSE))
         (RATIO (/ SIDE1 SIDE2))
         (DIFF (- HYP-SQ
                  (+ (* SIDE1 SIDE1)
                     (* SIDE2 SIDE2))))
         (DELTA (IF (PLUSP DIFF) DIFF (NEGATE DIFF))))
    (IF (<= DELTA (* 0.02 (* HYP-SQ)))
        RATIO)))
```

Figure 2.80: Program for Testing Whether a Hypotenuse and 2 Sides Form a Right Triangle

```
HT-MEMBER is a Set Membership operation.
  It determines whether or not ELEMENT is an element of the set
  STRUCTURE.
The set is implemented as a Hash Table.
  The Hash Table is implemented as an Array of buckets, indexed
  by hash code.
    The buckets are implemented as Ordered Lists.  They are ordered
    lexicographically.  The elements in the buckets are strings.
    An Ordered List Membership is used to determine
    whether or not ELEMENT is in the fetched bucket, BUCKET.

(DEFUN HT-MEMBER (STRUCTURE ELEMENT)
  (LET ((BUCKET (AREF STRUCTURE (HASH ELEMENT)))
    (LOOP DO
      (IF (NULL BUCKET) (RETURN NIL))
      (SETQ ENTRY (CAR BUCKET))
      (COND ((STRING> ENTRY ELEMENT) (RETURN NIL))
            ((EQ ENTRY ELEMENT) (RETURN T)))
      (SETQ BUCKET (CDR BUCKET)))))
```

Figure 2.81: Code for HT-MEMBER and Documentation Produced by Recognizer

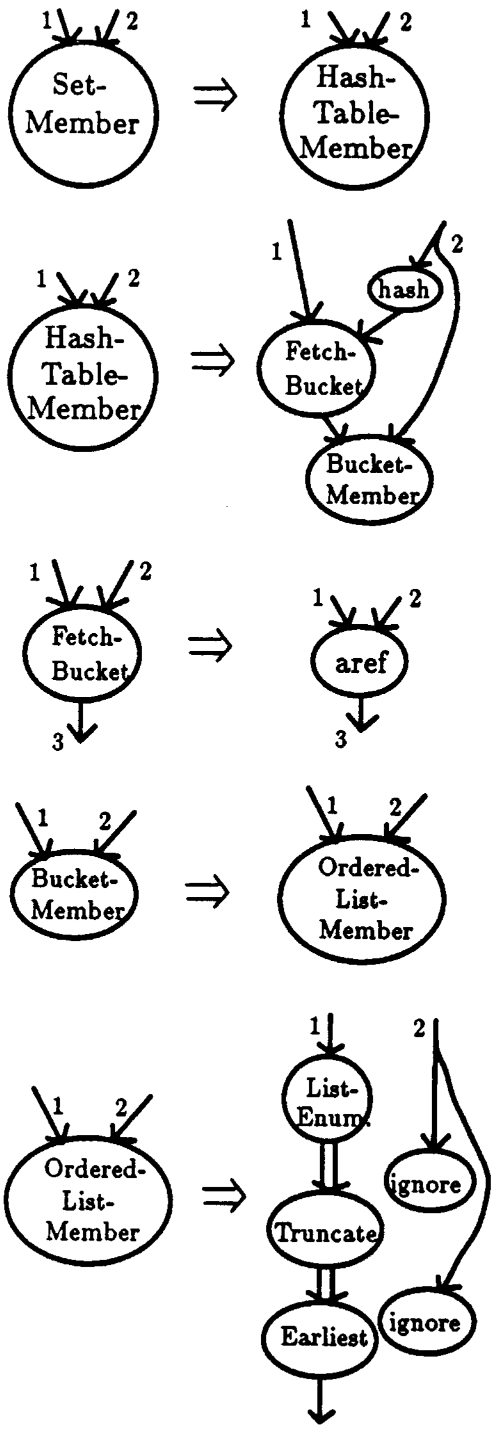
2.5 Examples Demonstrating All Capabilities

The examples shown in this section each make use of many of the Recognizer's capabilities. They are intended not only to show what the Recognizer can do, but also to help define its limits. This will be useful in discussing improvements and areas of future work.

The Introductory HT-MEMBER Example

Figure 2.82 shows the grammar used in analyzing the program given in the introduction to this chapter (and shown again in Figure 2.81 along with the documentation generated by the Recognizer). It displays most of the aspects of recognition already discussed, including the use of two partially evaluated functions in the Truncate and Earliest plans.

Many of the rules have annotations attached to them which uncover design decisions about



Annotations:

1. "The set is implemented as a Hash Table"
2. "determines whether or not [input 2] is an element of the set [input 1]"

Attribute-Transfer:

1. transfer all attrib's to corresponding attrib's

Attribute-Transfer:

1. Ce := ce of any of the nodes
2. success-ce := success-ce of Bucket-member
3. failure-ce := failure-ce of Bucket-member

Attribute-Transfer:

1. Ce := ce of "aref"

Annotations:

1. "The Hash Table is implemented as an Array of buckets, indexed by hash-code."

Attribute-Transfer:

1. transfer all attributes to corresponding attributes

Annotations:

1. "The buckets are implemented as Ordered Lists"
2. If ordering-predicate is "string>", then "They are ordered lexicographically."
3. "The elements in buckets are [type of domain of ordering predicate]"
4. "An Ordered List Membership is used to determine whether or not [input 2] is in the fetched bucket [input 1]"

Constraints:

1. all co-occur
2. Truncate's Predicate is a binary relation
3. Trunc's Pred is closed with respect to [input 2]
4. Earliest's Predicate is partially evaluated "equal"
5. Earliest's Pred. is closed w/ respect to [input 2]

Attribute-Transfer:

1. Failure-ce = [term-ce of List-Enum] + [term-ce of Trunc]
2. Success-ce := termination-ce of Earliest
3. Ordering-predicate := Truncate's predicate
4. Ce := Truncate's ce

Figure 2.82: Grammar for Analyzing HT-MEMBER

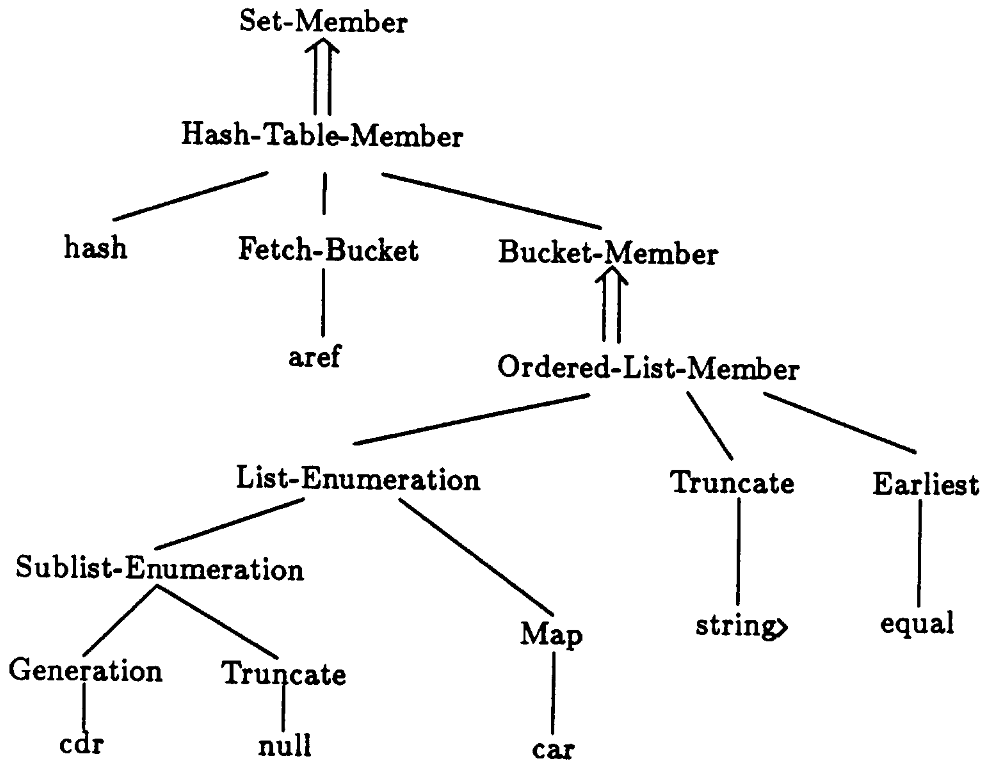
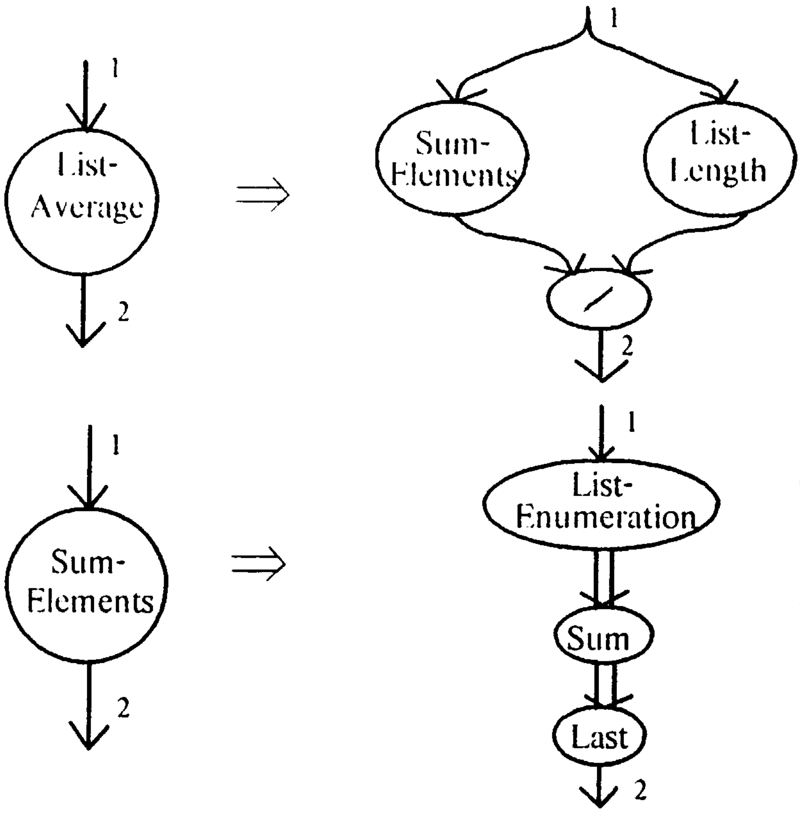


Figure 2.83: The Parse Tree for HT-MEMBER

how particular data objects are implemented. Note in particular that the type of elements in the ordered list is figured out by identifying the domain of the ordering predicate.

A further point to note is that in the attribute-transfer specifications for Ordered-List-Member, the failure-ce of the left-hand side non-terminal becomes the sum of the termination-ces of List-Enumeration and Truncate.

The parse tree for HT-MEMBER is shown in Figure 2.83. An upward double arrow (⇑) connecting two nodes of the tree signifies that the rule which expressed one in terms of the other was induced by an implementation overlay. For example, there is a double arrow from Hash-Table-Member to Set-Member. This means that the overlay Hash-Table-Member>Set-Member induced the rule which takes the nonterminal Set-Member to Hash-Table-Member.



Constraint:

1. List-Enumeration and Sum co-occur

Attribute-Transfer:

1. Cc := cc of List-Enumeration or Sum

Figure 2.84: Rules for List-Average and Sum-Elements

List Average

The *List-Average* plan (shown in Figure 2.84) shows the ability of the Recognizer to recover the design of a program even when parts of the implementations of two distinct abstract operations overlap. The List-Average cliché will be found in programs like the one shown in Figure 2.85.

However, a good programmer, realizing that the separate coding of the Cdr-Generations and loop terminations is inefficient, will optimize the code so that they are shared among the Sum-Elements and List-Length computations as in the code in Figure 2.86.

Even though the grammar for List-Average does not explicitly share the Cdr-Generation and the exit predicate, List-Average will be recognized. The parse tree of the program is not strictly hierarchical. (See Figure 2.87.) In it, there are two nodes, the exit predicate (NULL)

```
(DEFUN AVERAGE-LIST1 (L)
  (/ (SUM-UP-ELEMENTS L)
     (LIST-LENGTH L)))

(DEFUN SUM-UP-ELEMENTS (L)
  (LET ((SUM 0))
    (LOOP DO
      (COND ((NULL L) (RETURN SUM)))
      (SETQ SUM (+ (CAR L) SUM))
      (SETQ L (CDR L)))))

(DEFUN LIST-LENGTH (L)
  (LET ((I 0))
    (LOOP DO
      (COND ((NULL L) (RETURN I)))
      (SETQ I (1+ I))
      (SETQ L (CDR L)))))
```

Figure 2.85: Unoptimized Code Containing List-Average

```
(DEFUN AVERAGE-LIST2 (L)
  (LET ((SUM 0)
        (COUNTER 0))
    (LOOP DO
      (COND ((NULL L) (RETURN)))
      (SETQ COUNTER (1+ COUNTER))
      (SETQ SUM (+ (CAR L) SUM))
      (SETQ L (CDR L)))
    (/ SUM COUNTER)))
```

Figure 2.86: Optimized Code Containing List-Average

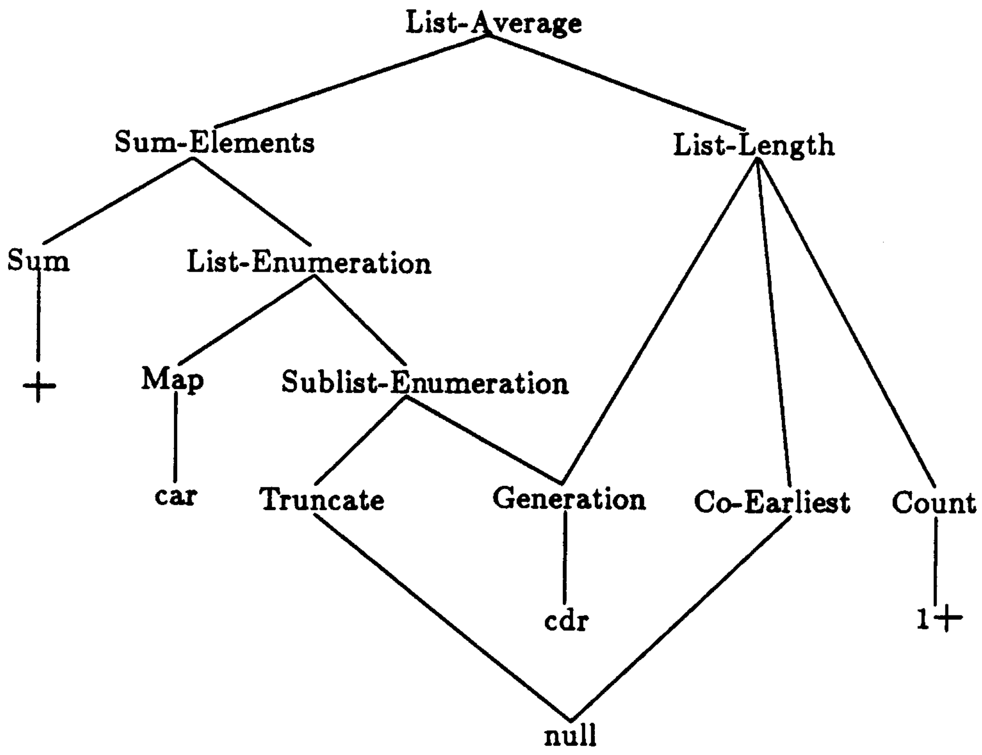
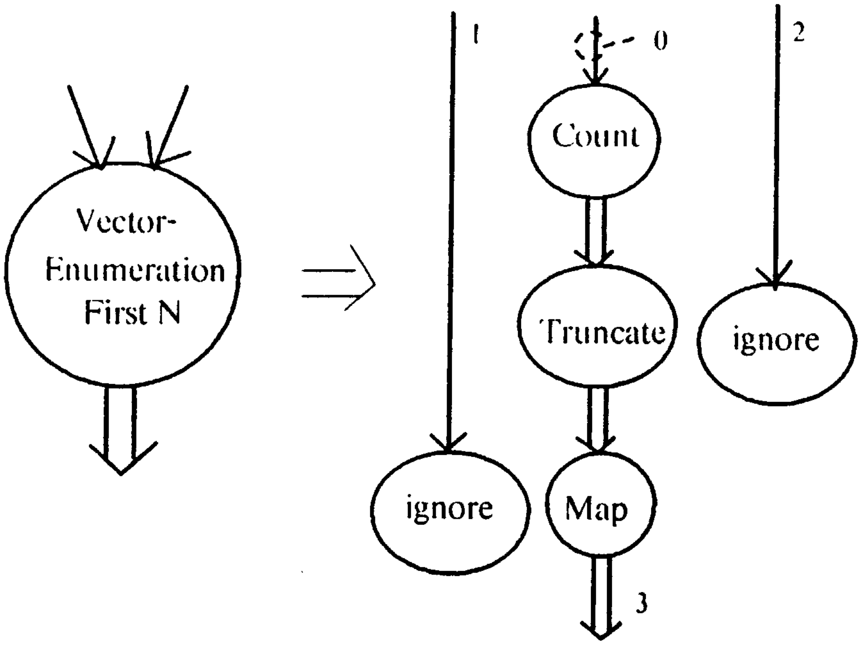


Figure 2.87: Parse Tree for AVERAGE-LIST2

and the Cdr-Generation, that are shared.

Arbitrary Composition of Loop Plans

Because the Recognizer may use any non-terminal as a start type of the grammar and because parses are started everywhere in the program's graph, the Recognizer is able to find basic loop plans in a program even when the program does not contain a specific higher-level cliché known to be implemented by the particular composition found in the program. For example, in the following function, the Recognizer will find the cliches: Sum (which accumulates the elements of a sequence by adding them together), Filter (using the predicate PLUSP), and Vector-Enumeration of the first N elements of the vector V, where V is implemented as an array.



Constraints:

1. Truncate's predicate is >=
2. >= is partially evaluated
3. >= is closed w/ respect to its second arg which receives data flow from <input 2>
4. Map's Function is "aref"
5. "Aref" is partially evaluated
6. "Aref" is closed w/ respect to its first arg which receives data flow from <input 1>
7. Count, Truncate, and Map co-occur

Attribute-Transfer:

1. Cc := cc of any node

Figure 2.88: Rule for a Vector Enumeration of the First N Elements of a Vector

```
(DEFUN POSITIVE-SUM (V N)
  (LET ((SUM 0) (I 0)
        (ELEMENT NIL))
    (LOOP DO
      (COND ((>= I N) (RETURN SUM)))
      (SETQ ELEMENT (AREF V I))
      (IF (PLUSP ELEMENT)
          (SETQ SUM (+ SUM ELEMENT)))
      (SETQ I (1+ I)))))
```

The rule for Vector Enumeration of the first N elements of a vector is shown in Figure 2.88. It is composed with a Filter and a Sum followed by a Last, as is shown in Figure 2.89.

The documentation generated for this function is:

```
POSITIVE-SUM is a Sum of a Filter of a Vector
Enumeration of the first N elements of V.
  The vector V is implemented as an array.
```

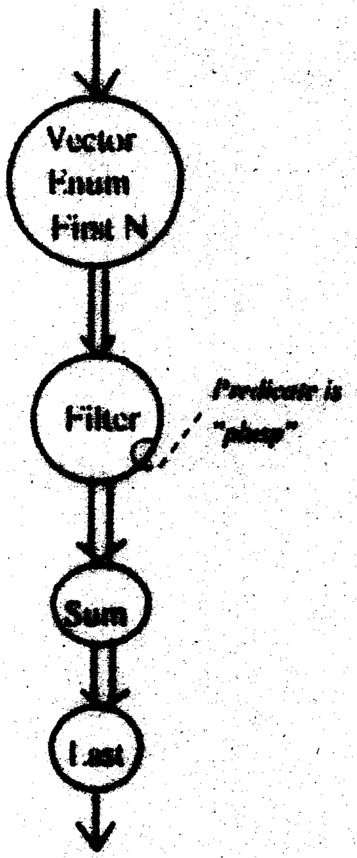


Figure 2.30: Temporal Abstraction of POSITIVE-SUM

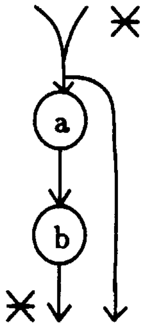
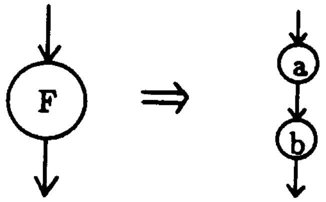


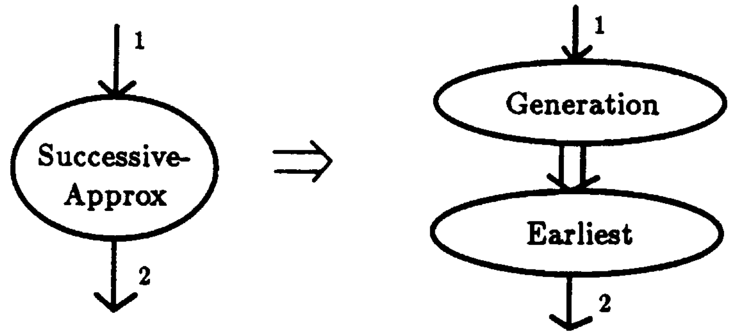
Figure 2.90: Flow Graph Having the Structural Form of a Generator

Waters' loop analysis algorithm ([41,42]) has this same ability to decompose a loop into the plans of which it is constructed. (The example just discussed was taken from [42].) The difference between Waters' analysis technique and that of the Recognizer's is that Waters groups together operations in a program into maximal segments based on control flow, while the Recognizer finds single operations that display the functionality of temporal fragments based on data flow. In Waters' system, arbitrary subgraphs may be bundled up into a loop plan, rather than just single operations. For example, the flow graph shown in Figure 2.90 has the form of a Generation. However, it will not be recognized as such because the Recognizer cannot group together the two operations **a** and **b** into the single composed operation $a \circ b$. In analyzing the code, Waters' algorithm would group these together as a Composition. In this way, the Generation plan would be recognized. The Recognizer must be extended in the future to combine the power of Waters' segmentation techniques with its existing capabilities. This may involve adding grammar rules like the following.



Constraint:
"a" and "b" co-occur
Attribute-Transfer:
F := composition of "a" and "b"

The next example (Square Root) shows how this may be useful.



Constraints:

1. Generation and Earliest co-occur
2. Gen's Function is an improvement function
3. Earliest's predicate is a criterion for convergence

Attribute-Transfer:

1. Ce := ce of either node
2. Improvement-Function := Gen's Function
3. Convergence-criterion := Earliest's Predicate

Figure 2.91: Rule for Successive-Approximation

Square-Root

The following code computes the square root of a number using Newton's approximation method. The flow graph for this program is shown in Figure 2.92.

```
(DEFUN MY-SQRT (N)
  (LET ((APPROX 1))
    (LOOP DO
      (COND ((< (ABS (- (* APPROX APPROX) N))
                0.0001)
             (RETURN APPROX))
            (T (SETQ APPROX
                     (/ (+ APPROX (/ N APPROX))
                        2.0)))))))
```

This function can be seen as an instance of the *Successive-Approximation* cliché in which a sequence of values are generated and then tested to see if they are close enough to the correct answer. The rule for Successive-Approximation is shown in Figure 2.91. An instance of it occurs in **MY-SQRT**. In this instance, the generating function for its Generation is the function for computing an improvement to the current approximation, i.e.,

```
(/ (+ APPROX (/ N APPROX)) 2.0).
```

Earliest's predicate is a criterion for convergence, i.e., that the square of the approximation be within an epsilon (0.0001) of the input number. In order for the Recognizer to recognize a

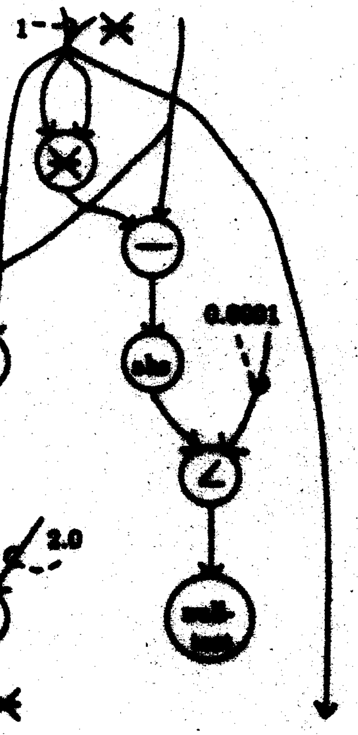


Figure 2.32: Flow Graph for MY-SQRT

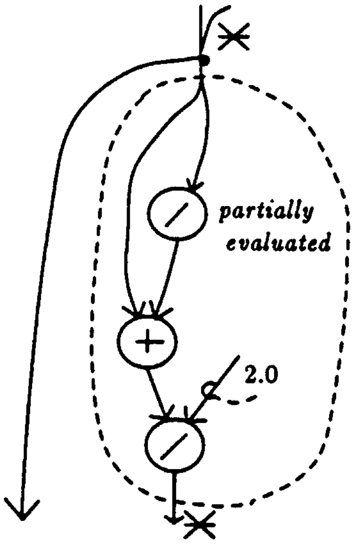


Figure 2.93: Part of the Flow Graph for MY-SQRT

Generation and an Earliest cliché (and therefore a Successive-Approximation), the grammar must contain specific rules for finding the improvement step and criterion for convergence. Otherwise, the Recognizer will not be able to bundle up the appropriate subgraph into a generating function or an exit predicate.

The Recognizer might be made more powerful by being able to recognize the structural form of an arbitrary subgraph as being that of a specific loop plan. This way, the subgraph shown in Figure 2.93 would be recognized as a particular generating function.

In order for this to be useful, however, the Recognizer must also be able to prove that the subgraph bundled up performs the appropriate function. The Generation in Successive-Approximation requires that the generating function produce a new value on each iteration, based on the previous value. The Generation of improvements and the exit predicate of the Earliest plan when used together must be proven to converge. The Recognizer, therefore, needs to incorporate help from outside sources, such as the user, a theorem prover, or systems which contain domain specific knowledge (e.g., Macsyma).

Chapter 3

Limitations and Future Work

There are quite a few areas of research opened by both the limitations and the capabilities of the Recognizer. The first part of this section discusses work which needs to be done to improve the Recognizer. The second part discusses some areas of research where the Recognizer may be applicable. The third describes how it may be incorporated into software development and maintenance tools and into a programming tutor.

3.1 Limitations of the Recognizer

The Recognizer, like most experimental systems, has much room for improvement. Many of its limitations have been touched on briefly throughout the earlier sections. These will be discussed in more detail in this section.

Types of Programs Analyzed

The Recognizer is able to recognize clichés in programs which contain nested expressions, conditionals, single- and multiple-exit loops, and some data structures. The Recognizer cannot handle any side effects, other than assignment to variables, nor can it analyze programs containing recursion or arbitrary data abstraction. Future work is required to allow the Recognizer to handle these features.

Side Effects

The only side effects which may occur in programs analyzed by the Recognizer are assignments to variables. This is because the Flow Analyzer has built-in knowledge about what it means for an assignment to occur. It is able to show the net data flow as a result of an assignment. However, in order to understand side effects, such as those which occur in the use of RPLACA and RPLACD, the Flow Analyzer must be extended to correctly represent the net data flow. (See [28,37].) This may involve its interaction with program recognition.

Data Abstraction

By using rules derived from implementation overlays, the Recognizer is able to recognize the use of certain standard data structures, such as sets, lists, and hash tables, and is able to explain how they are implemented in terms of each other. However, in order to analyze programs which use aggregate data structures, the Recognizer must be able to destructure the data objects. The problem is that plans which act on aggregate data structures take the data structure as input, but must decompose it in order to act on its parts. For example, the cliché *Push* acts on a stack which consists of a collection of values (the "base") and an index into them (the "lower"). The plan for Push is shown in Figure 3.1. If the stack were implemented as an array, for example, the subplans *Bump* and *Update* would be implemented as DECREMENT and ASET, respectively. In the plan, the input stack must be seen as two separate data objects. In the current system, there is no way to represent the destructuring and aggregation of the components of a data structure. However, the mechanisms for viewing programs from several points of view may be helpful in providing this capability in the future.

Recursion

The techniques employed in the analysis of loops are expected to be useful in analyzing programs containing recursion. The same types of feedback correspondences must be recorded between the inputs to a recursive program and the inputs to a recursive node. In addition, output correspondences must be maintained which show how the output of the program relates to the output of the recursive node. For example, the recursive program COPYLIST which copies a list is shown with its flow graph in Figure 3.2. (The correspondences are indicated by subscripted asterisks.)

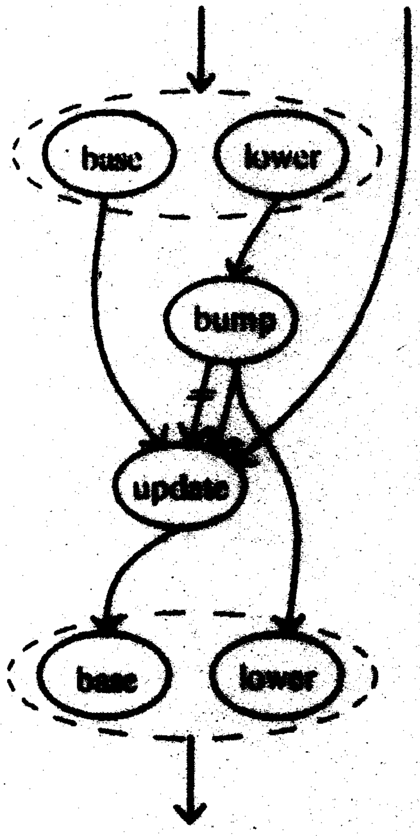
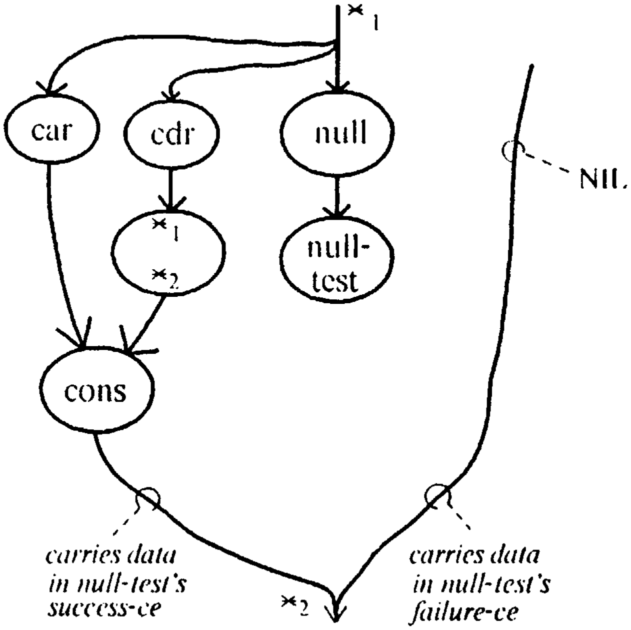


Figure 3.1: The Plan for an Implementation of Push

```
(DEFUN COPYLIST (L)
  (LABELS ((INTERNAL-COPYLIST (L)
             (COND ((NULL L) NIL)
                   (T (CONS (CAR L)
                            (INTERNAL-COPYLIST (CDR L)))))))
    (INTERNAL-COPYLIST L)))
```



*1
car
cdr
null
*1
*2
null-test
NIL
cons
carries data in null-test's success-ce
*2
carries data in null-test's failure-ce

Figure 3.2: COPYLIST Program and Flow Graph

A problem with recursion which doesn't arise in analyzing loops is that the arity of the recursive node in a program's flow graph might not match the arity of the node in a rule for a recursive cliché. For example, suppose the COPYLIST flow graph were being searched for in the code shown (along with its flow graph) in Figure 3.3. Because the arity of the recursive node in the COPYLIST flow graph doesn't match the arity of the recursive node in the COUNT-AND-COPYLIST flow graph, the COPYLIST cliché will not be found.

Taking the approach of removing the recursive node from the graph (which is how this problem is avoided in loops) is not helpful. The recursive node is what connects the rest of the program's graph following the recursive call to the part of the program's graph before the recursive call.

A possible solution to this problem of arity is to apply transformation rules which allow the recursive node to be seen in a different way. For example, Figure 3.4 gives two possible ways to view the recursive node in COUNT-AND-COPYLIST. The first will allow the Copylist cliché to be recognized. The effect the transformation has on the input graph is to separate subgraphs which do not depend on each other into two distinct operations: one counting and the other accumulating the elements of a list. The transformed program is shown in Figure 3.5. This transformation is similar to the type of analysis Waters ([41,42]) does in pulling augmentations out of a loop.

Additional study is also needed to discover the important control environments in the body of a recursive function and to determine the relationships among them.

Size of Programs Analyzed

The Recognizer is able to gain a deep understanding of small programs. It needs to be extended in many ways in order to be able to deal with larger programs. The present system was not written with efficiency as a main goal. There was a constant struggle against the complexity of the problem. Therefore, techniques were implemented in straightforward, rather than cleverly efficient ways. Although some optimizations (such as using dynamic programming and suspending and restarting parses) were introduced, there are other areas which need to be made more efficient.

The current system flattens the entire program graph by open-coding all subroutines (except recursive ones) inside the bodies of their callers during macro-expansion. However, selec-

```
(DEFUN COUNT-AND-COPYLIST (L)
  (LABELS ((INTERNAL-CC (L LENGTH)
             (COND ((NULL L) (PRINT LENGTH) NIL)
                   (T (CONS (CAR L)
                            (INTERNAL-CC
                              (CDR L)
                              (1+ LENGTH)))))))
    (INTERNAL-CC L 0)))
```

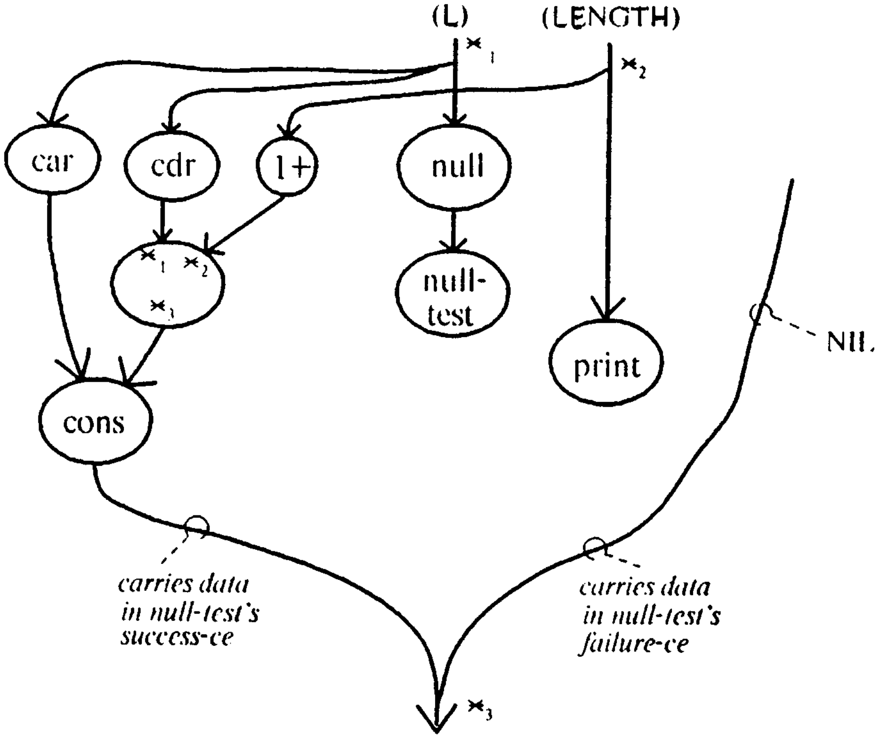

(L)
(LENGTH)
×1
×2
car
cdr
1+
null
×1
×2
×3
null-test
print
cons
NIL
carries data in null-test's success-ce
carries data in null-test's failure-ce
×3

Figure 3.3: COUNT-AND-COPYLIST Program and Flow Graph

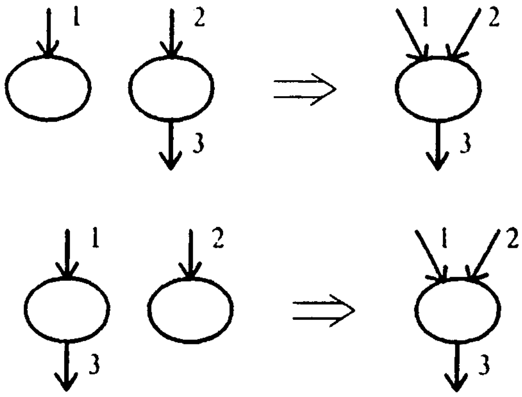


Figure 3.4: Transforming a Recursive Node

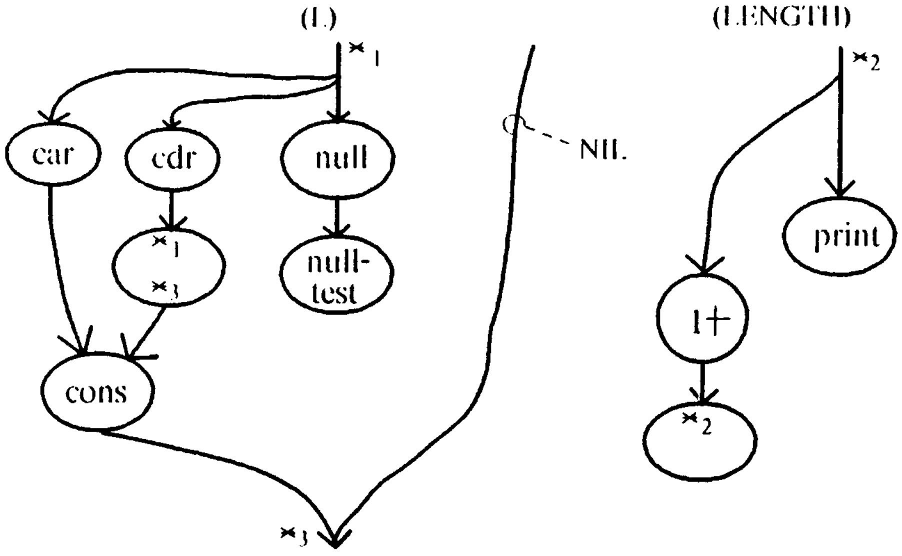


Figure 3.5: Transformed Flow Graph for COUNT-AND-COPYLIST

tive expansion may be done by translating each definition of a subroutine into a transformation rule. The rule's right-hand side would be a single node with the subroutine's name as its type and the plan for the body would be the rule's left-hand side. A program which calls the subroutine would have a node whose type is the subroutine's name. An expansion of this node into the subroutine's body may be performed if more parses were desired. Which subroutines to expand and when to expand them must be guided by a higher level control mechanism. This feature may be helpful when flattening an entire program is impractical because of size.

Another optimization is to provide heuristics to cut down on the number of parses generated. They also may be added to direct the parsing to specific locations in the graph where a certain cliché is more likely to be found or which may be ignored.

Other Psychologically Valid Models

The Recognizer uses parsing as a model of how people understand programs. There are other valid psychological models as well (e.g., [2,40,46]). People use a variety of other methods for determining what a program does and how it does it. For example, besides analyzing code, it helps to be able to "play" with it, i.e., run some test cases and see what it does. In addition, any documentation surrounding the code may be helpful. It may be useful to construct a model of how people monitor the behavior of code while trying it out or a model of how people glean information from documentation and connect the documentation with the parts of code that implement it.

In the future, it might be useful to combine systems based on several theoretical models in a hybrid system which more closely emulates the understanding process of people.

User Interface

The Recognizer's user interface may be enhanced by a variety of features, including the following.

- By providing automatic translation from a library cliché to a rule or from code which defines the cliché to a rule, entering rules into the grammar may be made easier.

- Rules which use constants may be given as input and the grammar may be automatically processed to incorporate the constants into the functions that use them. This would also involve automatically generating the transformation rules needed as a result.

- All possible sets of start types of the grammar may be automatically generated. Currently, trying to recognize non-terminals on all levels of the grammar is done by repeatedly invoking the Recognizer with a different set of start types each time. If all sets were generated automatically, the user wouldn't need to manually invoke the Recognizer more than once.
- If the Recognizer is used in a system where the code being analyzed is displayed for the user, parts of the code in which a cliché is found may be highlighted.

3.2 Relevant Areas of Research

Many research areas may benefit from the capabilities of the Recognizer. In particular, the transformation and parse restart mechanisms have the potential to be useful in a number of areas. They have been used in the Recognizer primarily in dealing with constants and partially evaluated functions. As has already been mentioned, they are expected to also be useful in the analysis of recursive programs and arbitrary data structures and in performing selective macro-expansion. In addition, the mechanism opens up new avenues of research into performing transformations in general, debugging, and learning new clichés as well as common bugs.

Transformations

The current system typically transforms a single node to another node (as in the case where "+" is seen as "1+" if one of the inputs comes from the constant 1). However, the transformation rules allow a nonterminal to be replaced with any subgraph. Transformations may allow more parses to be found by letting an implementation for one abstract operation be replaced by another implementation for that operation. For example, suppose an abstract operation (X) has two different implementations and suppose the cliché to be recognized were Y (as in Figure 3.6). If the program graph were the one shown in Figure 3.6, then Y would not be found. However, the Recognizer may let one implementation for X (-E-F-) be replaced by an alternative implementation (-B-C-), and thus allow Y to be found under the condition that X be implemented as -B-C- instead of -E-F-. The way that this sort of transformation is made possible is by letting each rule for X be both a transformation rule and a normal rule. Thus they may be run either forward or backward.

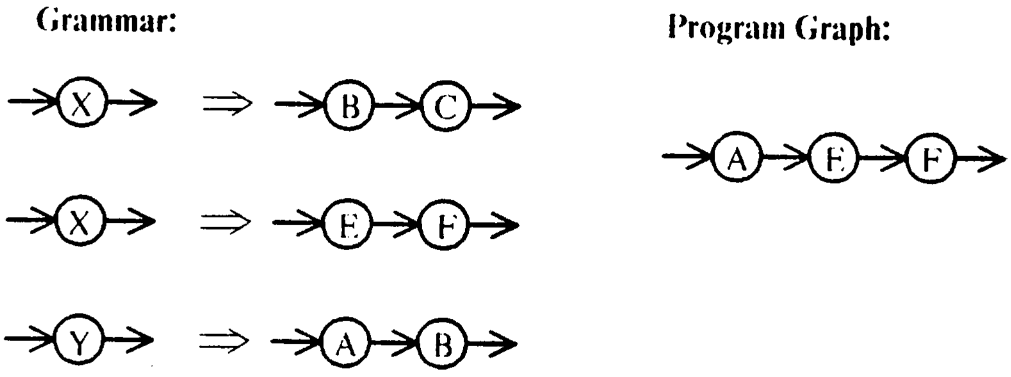


Figure 3.6: Grammar and Program Graph

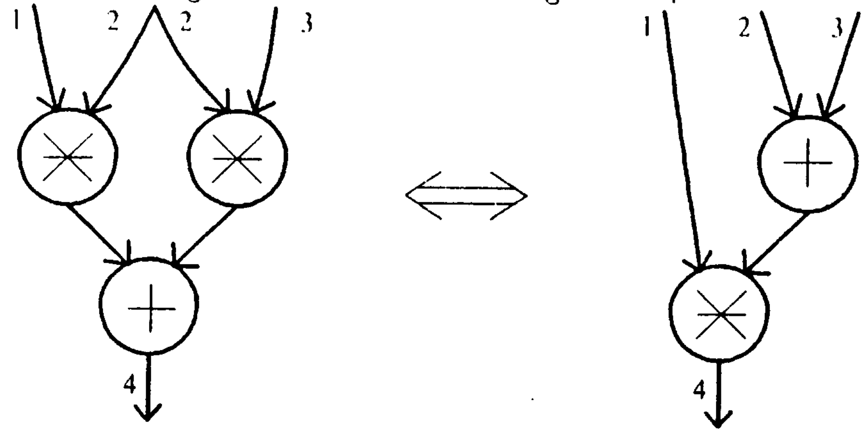


Figure 3.7: Transformation for the Distributive Law

Transformations are useful in embodying knowledge about a particular domain. For example, the transformation in Figure 3.7 gives the Recognizer the distributive law. (The double arrow (⇔) indicates that the rule may be used to transform the graph on either side into the graph on the other.) It may be used to find the Sum-of-Squares cliché in the following code. The flow graph for the code is given in Figure 3.8a. It must be transformed into the graph shown in Figure 3.8b in order for Sum-of-Squares to be found in it.

```
(DEFUN MAGNIFY-SUM-OF-SQUARES (M A B)
  (+ (* M (SQUARE A))
     (* M (SQUARE B))))
```

Since transformation and parse restart may occur anywhere in the graph and at any time after the subgraph to be transformed is found, a mechanism must be developed in the future

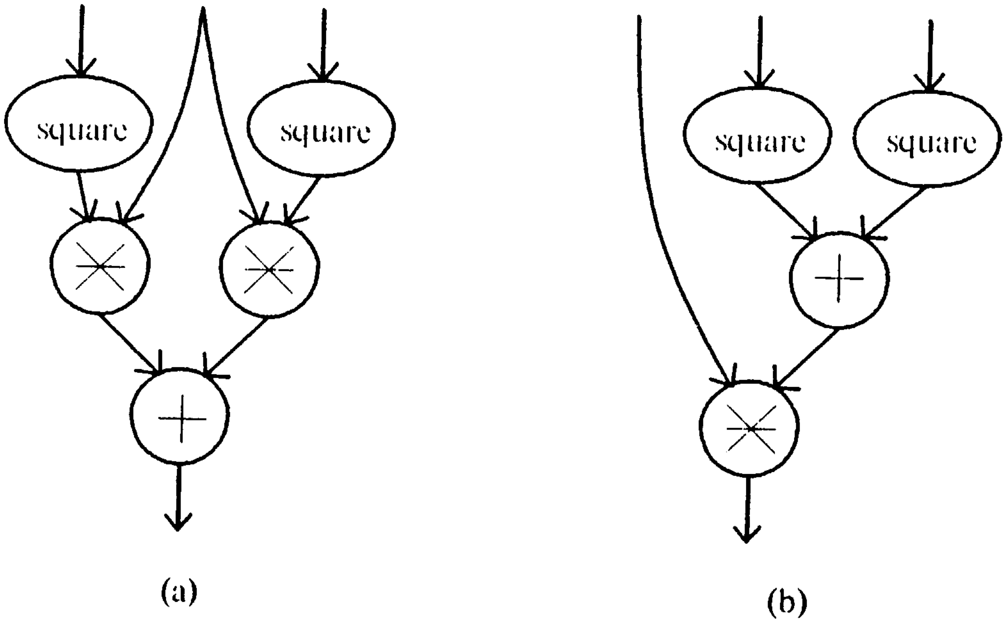


Figure 3.8: Flow Graphs for MAGNIFY-SUM-OF-SQUARES

to control when transformations take place and which parses should be resuscitated. It will probably be based on heuristics. The heuristics may be explicitly stated in a program or control diagram as is done in [5] (see section on Related Work). Being able to program the grammar and transformation rules in this way may be useful.

Debugging

The recognition system can be incorporated into a debugging tool, giving the debugger the ability to both find specific ***bug clichés*** and to do ***near-miss recognition*** [28]. Most current debugging systems (e.g., [17,34,35,38]) search the code for specific bug clichès. This is useful when bugs are treated as explicit entities which have information attached, such as advice or bug-fixes.

Rich points out the usefulness of near-miss recognition as another debugging technique. In near-miss recognition, clichés which are almost correct are pointed out as potential errors. For example, near-miss recognition can point out that a cliché can almost be recognized except that

- a particular operation was not performed or a subcliché was not recognized and therefore the larger one could not be found;
- the wrong operation was done on the correct data and the right thing was done with its result;
- arguments to a procedure or function were given in the wrong order;
- the arms of a conditional are switched;
- the return value of a piece of code is coming from the wrong place;
- a superfluous operation is performed within a program fragment in which a cliché was expected to occur.

An advantage of this type of error detection is that it is more flexible. It does not require having to enumerate all possible buggy versions of a cliché in order to be prepared to recognize and intelligently explain a bug. Error messages can be generated automatically based on the knowledge of the cliché's behavior and what is being left out or done improperly. This cuts down on the design cost of the system and gives it more flexibility in dealing with many different situations.

A number of near-misses can be recognized by collecting those parses which pass graph matching but fail some constraint. These parses may be suspended temporarily. A reason for being suspended may be associated with each of them. This would serve as a hypothesis about what the error is. In order to correct the error, heuristics or transformation rules may be employed. When they are applied, the suspended parses may restart. If the parses go on to succeed, then a reason for the bug (i.e., the reason for suspending the parse) may be given as well as how to fix it.

Program recognition facilitates automatic program modification to fix bugs not only in performing near-miss recognition, but also in using bug clichés. Bug clichés are "pre-debugged" in that their fix may be included in the information associated with them. This may be in the form of an transformation rule which gives the correct version of the cliché.

Shapiro's Sniffer [35] is an example of what is possible using program recognition in debugging. His cliché-finders do a kind of program recognition using exact match. Sniffer relies on the fact that the programs being debugged can be represented using a single cliché with little or no variation in the roles and subclichés. For larger algorithms, subclichés can be implemented in a variety of ways. This poses a problem for Sniffer, but it is very easy to deal with using flow graph grammars. Flow grammars are able to capture several implementations for the same higher level operation on any level of design. Thus, the generality of the cliché-finders would be increased by the more powerful parsing technique used by the recognition system.

Learning New Clichés

New clichés may be learned by a system which uses the Recognizer in two ways. In both, a top level specification for what is to be learned (i.e., what the new cliché is supposed to do) is given. First, near-misses and their reasons for not being quite right may be collected. Then a new cliché may be created which is the original (known) cliché with a slight modification based on the reasons why the cliché is a near-miss. Whatever part of the specification corresponds to the near-miss will be the specification for the new cliché.

The second way clichés may be learned is to give the Recognizer a program containing the new cliché. The Recognizer would identify as much of the program as possible and would match up the recognized structures with relevant parts of the specification. Any parts of the program not recognizable as a known cliché may be matched up with the remaining parts of the specification. This may require the use of a theorem prover to prove that an unrecognizable subgraph actually does what the corresponding specifications for it require. Rich and Waters first proposed this idea and gave a good example of it in [33]. See [15] for related work on learning rules about design by trying to understand as much of an example design as possible and then formulating a conjecture about the unrecognizable sections.

Learning Common Bugs

Bug clichés may be learned in the same way as correct clichés. Near-misses may be recorded along with the reason the correct cliché does not occur in the code. Alternatively, the Recognizer may be given explicit instances of bugs in code and any differences between the code and the cliché it is supposed to contain may be recorded as the cause of the bug.

Bugs may also be learned without being accompanied by a specification. During debugging, the restart mechanism allows hypotheses to be tested about what is wrong with a near-miss.

If a hypothesis holds true (i.e., by fixing it, the cliché is found), then a type of bug will have been found.

3.3 Applications of Program Recognition

Automatic program recognition has applications in at least two main areas. First, it makes possible a set of new AI-based software development and maintenance tools. Their effectiveness comes from their deep understanding of the programs to which they are applied. A second domain in which recognizing a broad range of programs is useful is computer-aided instruction. This section discusses the ways the Recognizer can be incorporated into software development and maintenance tools and into a programming tutor.

Intelligent Tools

An analysis system supporting program recognition will be an essential part of the next demonstration system of the Programmer's Apprentice (PA) project ([31,32,33,44]). The PA is an intelligent software development system designed to assist expert programmers in all aspects of programming. Many of the tools provided by the PA will benefit from the use of a program recognition module. This section points out some of the tools for which program recognition is expected to be useful.

Documentation

The ability to recognize familiar constructs in code allows automatic generation of documentation to explain these parts and how they fit together. This description will reflect a deep understanding of the program's behavior and structure, rather than simply giving the program's syntactic features, such as the number of arguments it has.

Verification

Rich [28] discusses the applicability of program recognition to verification. Clichés can be pre-verified in the sense that fragments which programmers use over and over usually have been tested and their behavior is predictable. Because of this, recognizing clichés in code can increase the programmer's confidence in the code. There is also ongoing work on formally pre-verifying clichés [30,29].

Translation

Translation is the process of converting a program in one language into another. This is done usually for the purposes of making the program more efficient (as does compilation) or making it more readable (as does source-to-source translation). Being able to recover the top-down design tree of a program coded in the source language facilitates the automatic rewriting of the design in the target language (see Waters [45]). Faust [12] shows the feasibility of this approach to source-to-source translation in order to relieve the burden of understanding on the part of the maintainer. He built a prototype system which takes COBOL programs, converts them to the plan representation, and then abstracts them onto a higher level by analyzing them in terms of Plan Building Methods (PBMs) [41,42]. The analyzed program is then converted into HIBOL, a high level language for data processing applications. Faust's system is limited by the special-purpose techniques it uses to recognize specific features of COBOL programs. The system would benefit from a general program recognizer which would broaden the class of programs that may be translated.

Maintenance

The key to being able to maintain a program is being able to understand it. Translation and documentation generation are two ways discussed so far wherein automatic program recognition can help increase a maintainer's understanding of code. Another way is that analysis based on program recognition can generate more informative explanations about what can be expected to happen if a certain change is made to the code. The explanations are in the programmer's vocabulary and relevant pieces of code may be pointed out.

Computer-Aided Instruction

Program understanding is essential in the domain of computer-aided instruction of programming. Besides simply giving a student pre-stored exercises to solve, a programming tutor must have an understanding of what the student's solution is supposed to achieve. It must be able to recognize whether or not the student's code does achieve that goal in order to check the correctness of it and debug it. An important part of this is being able to deal with great variability in acceptable solutions. The tutor should not be distracted or thrown off by extra activity, such as print statements, or by bugs which cause sections of code to be unrecognizable.

This section gives a description of how program recognition can be applied to computer-aided instruction which is more detailed than the discussions of other applications. This is because the tutor takes advantage of most of the capabilities of the Recognizer. Two of the most important are its ability to deal with broad classes of equivalent programs and to perform partial recognition when parts of the code are not familiar or are buggy. The tutor also uses some of the applications described earlier. A design for the tutor is given in this section to show more specifically the role the Recognizer would play in such an application.

Overview

Figure 3.9 gives a block diagram of a recognition-based tutoring system. The tutoring system interacts not only with the student, but also with the teacher who is developing the curriculum.

The ways the tutor interacts with the teacher are the following. The system allows the teacher to set up exercises for the student to solve. For each exercise, the teacher gives a problem statement and a grammar defining a top-down design of a typical solution. The grammar corresponding to the teacher's solution is mapped to the problem statement in the exercise bank.

The top-level rule given by the teacher simply maps an exercise name to a plan which implements its solution. However, most of the rules in the grammar specify clichés. These may be used in several of the teacher's solution designs. Rather than duplicating much of the grammar rules each time a solution grammar is given, the teacher may maintain a library of rules which specify clichés and which may be used in several solution designs.

This requires a bottom-up construction of the library. If the teacher adds a low-level cliché after defining a higher-level one which contains it, the library won't show the hierarchical

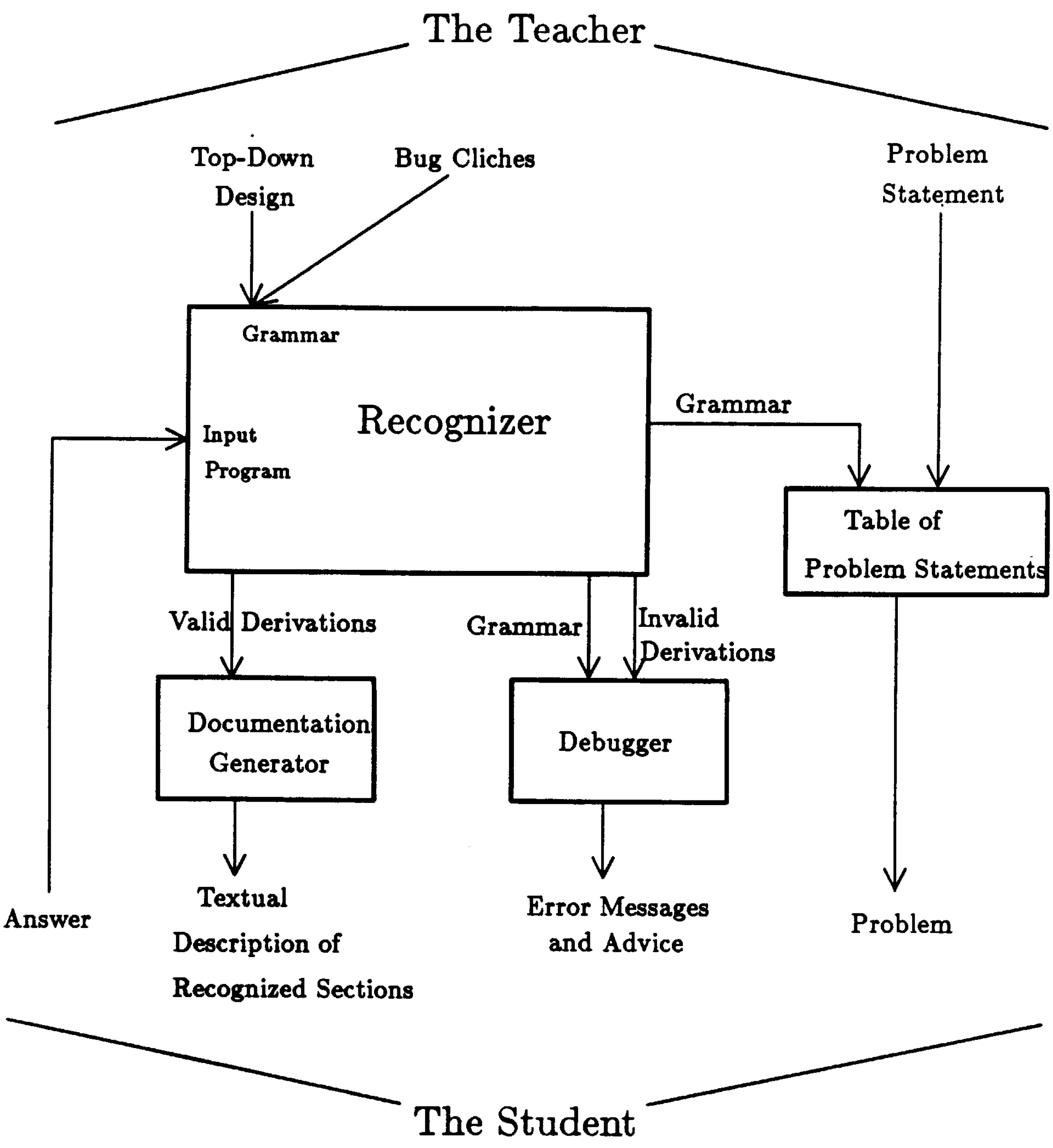


Figure 3.9: Organization of the Tutoring System

relationship between the two. In order for it to do this, every time a cliché is defined, each of the existing clichés should be examined to see if the new cliché can be recognized in them.

The teacher is able to automatically define not only programming clichés, but also bug clichés which contain knowledge about specific programming errors. These bug clichés are used by the debugger in order to generate error messages and advice. More will be said about this in the section on the debugger.

The student interacts with the system in the following ways. The tutor gives an exercise to the student who provides a program as the answer. The system then checks the student's code by trying to derive it using the grammar describing the teacher's design. It gives a textual description of the code based on the clichés recognized in it, as well as error messages and advice on how to fix the code.

Features

Several of the key strengths of the Recognizer are valuable to the tutor in dealing with student programs. In particular, the canonical representation for programs allows student programs to vary considerably in terms of style, control constructs used, ways of binding variables, and ways of breaking the code into subroutines. Given a good enough library (something which is easy to extend and improve), the Recognizer can deal with student programs which also display great variability in types of data representations used and implementations of algorithms.

Another feature of the Recognizer which is crucial to the tutoring system is the ability to perform partial recognition. Starting multiple parses allows parts of the program to be skipped over and ignored. The student may have irrelevant details thrown in, but these will not distract the Recognizer. For instance, a student's code may have print statements sprinkled throughout. It may also do computations on the side, such as counting the number of iterations so far for debugging purposes or for checking for error conditions in input data. The system can still recognize the correct solution in the midst of the extra code. Note that the extra activity must be done in parallel with the correct cliché operations. If extra operations were spliced into the data flow in the middle of a cliché, the code fragment no longer is recognizable as that particular cliché, and rightly so, since the data flow is disrupted.

The Debugger

A key component in the tutoring system is the debugger. In tutoring, the process of finding bugs, describing them, and either suggesting corrections or guiding students to a correction is an important part of the teaching process. The application of the Recognizer to debugging in general has already been discussed. The main capability the Recognizer gives to the debugger is being able to understand the code and the intended design of the code and to point out bugs in an informative way.

Knowing the clichés involved and how they fit together in the design of the program for a typical solution helps to better explain how the student's code deviates from the correct solution when a bug is found in the student's code. The student's error is localized in a specific cliché or role rather than simply saying that the code failed to match the correct solution altogether. Since the error is localized, it is easier for the system to give a more pertinent error message and to suggest a fix.

Chapter 4

Related Work

This section discusses related efforts primarily in the area of program understanding through the recognition of familiar features. Other work in which attributed graph grammars are used in analysis is also discussed. The main differences between the Recognizer's approach to program understanding and those discussed here are in the types of familiar patterns recognized, the way programs are represented, and the recognition technique used.

In contrast to the language-independent, canonical representation of programs which the Recognizer uses, many other systems use a programming-language based representation. This restricts them in the variability and complexity of fragments and structures that can be recognized because they end up wrestling with all possible syntactic variations and doing many transformations to twist the program into a canonical form. As programs get larger and more complex, the variety of ways that higher level operations can be expressed in code and the number of transformations needed to handle all cases grows exponentially. The problem of program recognition becomes one of trying to canonicalize the syntax of a program rather than focusing on its higher level features.

Furthermore, while the Recognizer uses an algorithmic technique, most of the other approaches rely on heuristics to discover characteristics of the program code. The methods used do not always guarantee that the structures will be found. If found, there is no higher level description of how they fit together.

The first section discusses systems that were developed in the context of program tutoring. The second section is concerned with general program understanding systems. The final section deals with related work in the area of using graph grammars to analyze graphical representations of images and conventional control flow graphs.

Tutoring Systems

The MENO-II system of **Soloway** [38] translates a student's program into an annotated tree whose leaves are variables and constants and whose nodes are operations. The annotations explain what roles the variables play in the program. This information is obtained by looking for certain kinds of statements in the code. For example, a statement like "SUM := SUM + NEW;" in a while loop is annotated as a *running total assignment* statement. A cliché in the Recognizer embodies a more general view of a piece of programming knowledge than the specific features which MENO-II tries to recognize. In order to extend MENO-II, increasingly more details need to be searched for as programs being analyzed become more complex.

MENO-II contains a component, called the BUG-FINDER, which uses pattern-matching on the parse tree to recognize in it certain sets of functional characteristics of the program. These sets of characteristics are also called *plans* in MENO-II but are more syntactic in nature than those of the Plan Calculus. Once the BUG-FINDER matches as much of the tree with known plans as it can, it looks in its bug catalog for common buggy ways of implementing those plans.

One of MENO-II's weaknesses is that the system is only able to find localized bugs in the code. It has no idea what the overall code is supposed to do or how it fails to do this. Neither does it know how to explain a bug in the context of the overall code. This would be avoided if the patterns to be recognized were more general and if they were stored in such a way that the interactions among them were made explicit. This is done in the Recognizer by storing the collection of clichés in the form of a grammar.

MENO-II relies on being able to look at the code and figure out what plans the student was trying to use. Most of the plan must be present in the code and must be almost correct for it to be able to pick up small deviations from it. In contrast, giving the Recognizer a grammar (e.g., having a teacher give the clichés involved in a correct solution) lets the Recognizer know ahead of time what higher level cliché should be recognized. This is important when dealing with possibly buggy code. In tutoring, it is especially important that the recognition system will not:

- get hung up if no plan is recognizable in the code (it must be obstinate in trying to find near-misses);
- say code is correct even though the code solves a problem different from that given in the problem statement (i.e., the code is a correct implementation of the wrong solution);
- work on perfecting the wrong plan by recognizing an almost correct plan but one which won't solve the problem even if it were correct.

Johnson and **Soloway**'s PROUST [17] tries to avoid MENO-II's failings by working bottom-up to recognize patterns and top-down from the programmer's intentions. This idea of guiding the analysis with information about the programmer's goals and how they may be achieved in the program has been adopted, not only by Johnson and Soloway, but also by Ruth [34], Miller [23] and Genesereth [14]. Rich [28] also discusses how bottom-up analysis by cliché recognition can be complemented by top-down analysis based on a description of the problem.

PROUST is given an informal description of the program's desired behavior. It then builds all possible goal structures based on knowledge it has about the relationships between goals and computational patterns and about how goals can be combined. It looks up the typical patterns which implement the goals and tries to recognize at least one in the student's code. If none fit, the system looks for buggy versions of the patterns in the code. The buggy versions are pre-stored in a bug catalog along with information about misconceptions a student may have which caused it.

In a tutoring system which uses the Recognizer, the teacher would give a top-down design of the program. This is the design that the student (hopefully) intends. This is not enough, however, to handle all the ways student programs may vary, especially for large programs. Problems come in when there are several possible goal structures.

Being able to work both top-down and bottom-up is PROUST's greatest asset. However, it is limited in that the standard programming structures it recognizes are textual in nature. This restricts their generality.

Ruth's system [34], like PROUST, is given a task description and the code to be analyzed which is supposed to perform the task. It tries to deduce the intended algorithm to achieve the task. Finding the intended algorithm is done by matching the code against several implementation patterns which the system knows about. The implementation patterns are in the form of a set of characteristics about the program (e.g. the number of iterations a loop performs and the predicate in a loop's exit test conditional).

During analysis, the code is checked for these characteristics. The entire program must be matched to an algorithm implementation pattern for the analysis to work. This is to be contrasted with the partial recognition performed by the Recognizer. Partial recognition enables understanding of as much of the code as possible even though clichés may be surrounded by unrecognizable code or extra computations.

Ruth's approach is the closest to the Recognizer's technique in that it uses a grammar to describe a class of programs and then tries to parse programs using that grammar. A key difference is that his system treats programs as textual entities. He canonicalizes the code by applying rewrite rules to it and then trying to parse it as a string. The Recognizer's graph representation is easier to canonicalize and also is inherently more abstract than code.

In debugging, Ruth's technique for finding errors is based on heuristics. His system checks for specific errors. For instance, when expression equivalence is not established, it checks if the expressions differ by a constant or by a sign. Ruth's system suffers from a problem similar to MENO-II in that it does only local analysis. It takes the task description and checks for pieces of the code which achieve the description's goal. It has no knowledge of the way program parts interact. It assumes they are independent of each other and that the program's statements are totally ordered (by control flow as well as data flow), rather than partially ordered (by data dependencies only).

Murray [26,25] presents an automatic program debugging system called Talus which uses a combination of heuristic and formal methods to debug Lisp programs. The program is parsed into an abstract frame representation in which functions are represented as *E-frames.* The slots of the E-frames contain information about the program. For example, some of the slots contain the type of recursion (e.g., list, tree, or number), the termination criteria, what recursive calls are made and under which conditions, and the data types of the inputs and outputs.

This representation helps abstract away from the syntactic code structure by extracting semantic features from the program, allowing greater syntactic variability in the acceptable student solutions. The variability is limited somewhat, however, by the fact that Talus requires that the student's code be broken up into subroutines in the same way as the model code it compares it to, in order for Talus to be able to debug it. Talus must use transformations to specify which functional decompositions are equivalent to that of the model program. This becomes a problem as the program being analyzed gets larger since the number of ways to break up the program into subroutines increases exponentially.

Another problem with Talus's representation is that because the characteristics of the code are represented by being listed explicitly in the E-frame, the representation is cumbersome and verbose. This too becomes increasingly harder to deal with as the programs being analyzed become more complex. The representation used by the Recognizer is more concise. It is easy to extract information from it, whether it is stored explicitly in attributes and labels or implicitly in the structure of the graph.

Talus has a library of stored algorithms. These are represented as a collection of model functions which carry out the algorithm. The functions are represented by E-frames. In analyzing the code, Talus looks for a stored algorithm that best matches the student's possibly buggy algorithm. It uses a best first search to perform the partial matching between stored functions and student functions. Similarity is measured by how closely the student's functions are mapped to the stored functions.

Heuristics are then used, based on any dissimilarities, to form conjectures about where bugs are located and how they should be corrected. Formal methods, such as symbolic evaluation, case analysis, and theorem proving, are then used to verify or reject these conjectures. The heuristics used by Talus rely on finding localized dissimilarities between the stored function and the student's function.

Talus is interesting in that it uses heuristics to locate specific, relatively small parts of the program where it may be useful to apply some formal method. This report has pointed out some places in which the effectiveness of the Recognizer may be improved by similarly incorporating other techniques, such as theorem proving, into the recognition process.

The LAURA system of **Adam** and **Laurent** [1] represents programs as graphs, thereby allowing syntactic variability. However, the graph representation differs from the Plan Calculus used by the Recognizer in that nodes represent assignments, tests, inputs, and outputs, rather than simply operations, and arcs represent only control flow. This means that data flow is represented implicitly in the graph structure. Because of this implicit representation of data flow, the system must rely on the use of program transformations to "standardize" the data flow. The Plan Calculus representation of programs shows net data flow explicitly, making these transformations unnecessary.

The system debugs a program by comparing it to a correct implementation, called the *program model* of the algorithm which the program is intended to achieve. This comparison is done using a heuristic approach. Since nodes are really statements of the program, the graph matching is essentially statement-to-statement matching. The system works best for statements that are algebraic or arithmetic expressions because they can be normalized by unifying variable names, reducing sums and products, and placing their terms in a specific order.

LAURA performs near-miss recognition in that when a slight deviation exists between the program model and the student's solution, the difference is corrected. If the corrected program subsequently matches the program model, then the deviation is reported as an error. LAURA is limited in that it deals with the syntactic differences between the model and the student program. The errors found are low level and localized. LAURA's error detection and correction capability could be made stronger by using a representation for programs similar to the Plan Calculus.

Laubsch and **Eisenstadt** [19] use a variation of the Plan Calculus to determine the effects of a program and then to compare what it achieves with what it was supposed to achieve. Their system differs from the Recognizer in the technique used to do recognition. Plans have effect descriptions attached to them so that when they are recognized, the description can be attributed to the code. Symbolic evaluation helps generate effect descriptions for unrecognized parts of the code. The actual matching of what the program should do and what it does is performed in terms of these descriptions. The problem of program recognition has been transformed into the matching of effect descriptions, i.e., the equivalences of formulas, which is in general extremely hard.

Other Program Understanding Systems

Lukey [20] has developed a theory of program understanding and debugging. His proposed method of analysis is to segment the program into *chunks* which are manageable units of code. (A loop is an example of a chunk.) He then describes the flow of information (or interface) between the chunks. Next, he looks for debugging clues. These are based on constraints which rational, correct programs obey and which are violated by buggy programs. Lastly, assertions are made which describe the values of the output variables of each chunk. These assertions are generated in two ways: schema recognition and symbolic evaluation.

Schema recognition associates a known pattern of statements to a description of their effects on the value of variables involved. The schema recognition process requires that the specific groups of statements making up the schema be localized. The statements cannot be scattered throughout the program or in a different (but equivalent) execution order. It also uses hints to recognize a schema, such as mnemonic variable names which are commonly used in the instantiation of a schema. This schema recognition is not performed as generally or algorithmically as in program recognition via graph parsing.

Lukey proposes that debugging be performed using two methods. The first, *tentative debugging*, is based on finding debugging clues. The proposed system finds suspicious looking code and tells the user that there is something wrong with it, but it can't tell what. When irrational code is found, special transformations are done which are not based on a deep understanding of the program. The second way debugging is carried out is by comparing the description of the program with its specification. Any discrepancies are pointed out as bugs. Since the description is based on assertions derived by recognizing specific patterns and by symbolic evaluation, it can say what to do to correct a pattern or an expression. It cannot, however, explain the bug in the context of the program or what effect the bug has overall, since it doesn't understand the program. It can only say how to treat a symptom of the bug.

Fickas and **Brooks** [13] proposed a system for program understanding based on recognition. The system uses *description trees*, hierarchical structures generated by programmers during the design process. In analysis, the program is broken down into *program building blocks* (pbbs). Implementation plans decompose non-primitive pbbs into smaller pbbs which are linked together by data flow and teleology. This is analogous to what grammar rules do for the Recognizer.

A task specification is given to the analyzer as well as the code in order to guide the understanding process. Hypotheses about the code are formed and verified. The code is searched for pbbs. They use a language-independent and general representation for clichés which allows them to be found in different forms in the code. Their technique for finding the pbbs, however, is very different from the Recognizer's algorithmic parsing method. The code is searched for certain *distinctive features* of a pbb to narrow down the search for matching pbbs. A typical distinctive feature is the interchanging of two elements in an array which hints that a sort is being performed. While a hypothesis is being verified, other outstanding clues (called *beacons*) may be found which suggest the existence of other pbbs and therefore create, modify, and refine other hypotheses about the code. This system, like the Recognizer, allows partial understanding. However, the Recognizer's technique is more algorithmic and systematic.

Lutz [21,22] is doing work very much related to that presented in this report. He also uses the Plan Calculus representation for programs. He has developed and implemented a graph parsing algorithm which may be applied to parsing plans. The algorithm is based on the chart parsing technique for strings, generalized to parsing flow graphs.

Related Work using Attributed Graph Grammars

Graph grammars are used extensively in pattern analysis work. This section will discuss the work of Bunke, which is particularly relevant to the techniques used by the Recognizer. It will also describe related research in the area of analyzing control flow graphs using graph grammars.

Rather than parsing, **Bunke** ([7,5,6]) uses context-sensitive graph grammars to transform an input image, such as a circuit diagram, into a graph which describes it. The description generated is in terms of characteristics of the graph, such as symbols, connections, and end-points of connection lines. The grammars are used to interpret hand-drawn circuits and flow charts.

Bunke's work is related to the Recognizer in two ways. The first is that graphs are augmented with attributes. When a rule is applied in the derivation of a right-hand side graph from a left-hand side graph, an *applicability predicate* is applied which expresses constraints on the nodes and edges of the left-hand side. The applicability predicate is analogous to the grammar rules' constraints on attributes in the Recognizer. Furthermore, the productions in Bunke's system have attribute transfer functions as do the rules of the Recognizer. These functions specify the values that the attributes of nodes and edges in the right-hand side become when the right-hand side is embedded in the host graph.

The second way that Bunke's work is related to ours is that his grammar has special rules which may be used to correct distortions in the input image which result from being drawn by hand. For example, there are special productions in the grammar for closing gaps in lines and for adding or removing lines or solder dots. These rules are analogous to the Recognizer's transformation rules.

The grammar is *programmed* in that there is an explicitly defined order in which the productions are to be applied. This is specified by the user in a *control diagram.* The use of a programmable higher level mechanism has the advantage of being easier to control. On the other hand, one advantage the Recognizer has over Bunke's is that the error correction and parsing are synergistically combined. While parsing is going on, possible places that the input graph could be seen in a different way are being looked for. If any of these places are found, the section of graph in question can be transformed and parses that depend on seeing the graph in a different way may complete successfully. Thus, there is only one parsing pass.

Farrow, Kennedy, and **Zucconi** ([11,18]) present a *semi-structured flow graph grammar* which can be used to analyze the control flow graph of a restricted class of programs for the purposes of compiler optimization of code. The grammar may be used to derive structured programs containing the standard control structures: sequential statements, conditional statements, while-loops, repeat-until loops, and multiple-exit loops.

They also provide a parsing algorithm which runs in time linear in the size of the graph. This algorithm is able to take advantage of the simplicity of the grammar.

Kennedy and Zucconi discuss how the semi-structured grammar may be augmented by attributes to yield a number of applications in the area of global flow algorithms and graph-based optimization algorithms.

Appendix A

The Constraint Sublanguage

The constraint sublanguage is used to describe the constraints that must hold for a parse to be valid. Some of the forms of this language are used to access attributes of the nodes, ports, and edges of flow graphs. Others are used as predicates which test properties of these attributes.

In addition, there are functions which map a node or port which is in a grammar rule with the node or port (respectively) in the graph parsed that matched with the rule's node or port. These functions are:

- **n>** *node-name* – gives the node in the graph parsed which matched with the node having the label *node-name* in the rhs of this rule.

- **pl>** *port-spec* – (where *port-spec* is an ordered pair of the form "(node-name port-label)" which specifies a particular port on a node in the rule) gives the port in the graph parsed which matched with the port specified by port-spec. If *port-spec* specifies a port on the lhs node, then give the port in the graph parsed which connected to this lhs port when the rhs was reduced to the lhs. (See for example the constraints on the rule for Absolute Value.)

- **p>** *port-spec* – *port-spec* always specifies a port on the lhs node of the rule. This function returns all of the ports in the input graph parsed which connected to this lhs port when the rhs was reduced to the lhs.

- **nt-n>** *node-name* – used only when *node-name* is the name of a nonterminal in the rhs of the rule. This allows access to a nonterminal's attributes which were computed via attribute-transfer.

- **nt-p>** *port-spec* – (where *port-spec* is as above) used only when the node on which the port is located is a nonterminal in the rule's rhs. This also allows access to the attributes of a port on a nonterminal when the attributes were computed using attribute-transfer.

- **input-name>** *port* – used in generating documentation, this either gives the variable name or constant type associated with *port* or generates an s-expression to describe how data flow coming from this port was produced.

The Attribute Accessors

The following forms access attributes of structures within a flow graph.

- **control-env** *node* – gives the control environment of *node.*

- **success-ce** *split-node* – gives the success control environment of *split-node.*

- **failure-ce** *split-node* – gives the failure control environment of *split-node.*

- **innermost-loop** *ce* – gives the loop control flow information of the innermost loop containing the control environment *ce.* The loop information tells in which control environment the loop feeds back and which control environment surrounds the loop.

- **feedback-ce** *ce* – finds the loop control flow information of the innermost loop containing the control environment *ce* and then returns the control environment in which the control of this loop feeds back to the beginning of the loop.

- **outside-ce** *ce* – finds the loop control flow information of the innermost loop containing *ce* and then returns the control environment into which the control of this loop exits.

- **termination-ce** *node* – *node* is a nonterminal which represents a basic loop plan that has been recognized in a program. This returns the control environment in which the loop is terminated.

- **continuation-ce** *node* – *node* is a nonterminal which represents a basic loop plan that has been recognized in a program. This returns the control environment in which the loop continues to be executed. This is the failure-ce of the exit test. It differs from the feedback-ce of the loop in that there may be more than one continuation-env in a loop (if there are more exits), while there is only one feedback-ce (which is the lowest continuation-env).

- **ce-from** *source-port sink-port* – returns the control environment in which the edge from *source-port* to *sink-port* carries data flow.

- **ce-used-in** *port* – returns the control environment in which the data coming into *port* is being used. This is the control environment of the node containing *port*.

- **function-info** *node* – This returns information about the function that *node* represents. This information includes the type of the function, which of the function's inputs is the function closed with respect to, and which ports these inputs are getting data flow from. (The inputs that the function is closed with respect to are receiving data flow from loop constants.)

- **predicate-info** *node* – This is similar to function-info, but the function involved is a predicate.

- **init-value** *node* – *node* is a nonterminal which represents a basic loop plan that has been recognized in a program, e.g., Accumulation. This returns the initial value that the loop plan takes as input.

- **closed-wrt** *function-info integer* – *function-info* is the type and closed-wrt information that a node has about the function (or predicate) it represents. *Integer* is a number of an input to the function. Closed-wrt looks up this input in the function-info's closed-wrt information and returns the port that is sending data flow to the input.

- **source-type** *source-port* – gives the source type of *source-port* (e.g., NIL, 0, 2) if the data from source-port is a constant.

Predicates

The following forms are predicates that apply to attributes and structures contained in flow graphs.

- **ce-le** *ce1 ce2* – returns T if operations in the control environment *ce1* are executed the same number of times as or less often than operations in the control environment *ce2*.

- **ce-lt** *ce1 ce2* – returns T if operations in the control environment *ce1* are executed less often than operations in the control environment *ce2.*

- **ce-equal** *ce1 ce2* – returns T if operations in the control environment *ce1* are executed the same number of times as operations in the control environment *ce2.*

- **co-occur** *node1 node2* – returns T if node1 and node2 have the same control environment.

- **feeds-back** *port1 port2* – returns T if there is a feedback arc between *port1* and *port2.*

- **in-a-loop** *node* – returns T if *node* is in a loop's body.

- **loop-constant** *port loop* – returns T if data coming into *port* is coming from a "loop constant" which means that the data doesn't change over the iterations of *loop.* (This is the case when it is coming from some node outside *loop.*)

- **any-source** *port* – returns T if data coming from *port* is constant.

- **partially-evaluated** *node* – returns T if *node* represents an operation that is closed with respect to any inputs.

- **exit-predicate** *split-node* – returns T if the success-ce of *node* is ce-le the outside-ce of the innermost loop containing *node*'s ce AND if the feedback-ce of this loop is ce-le the failure-ce of *node.*

- **source?** *port source-type* – returns T if *port*'s source type is *source-type.*

Appendix B

The Grammar

This appendix contains the grammar which the Recognizer uses. It is created by manually translating the cliché library into rules.

Syntax of the Grammar Rules

Grammar rules are defined using **defrule**. The form of the rule is as follows:

```
(defrule <name-of-rule> <name-of-lhs-node>
  <rhs-graph-spec>
  <mapping>
  [optional:
   :st-thrus <straight-through edge specs>
   :node-type-constraints <constraints on node types>
   :constraints <constraints on graph matching rhs>
   :att-transfer-specs <attribute transfer specs>
   :doc <documentation string>
   :implementation <documentation string>
   :transformation <NIL/ TRANSFORMATION-ONLY/ BOTH>
   :grammars <list of grammar names>])
```

- **name-of-rule** – a string made up of the node type of left hand side, followed by a ">", followed by the name of the right hand side graph. For example, if a rule has the name

"A>some-graph" then the rule has a nonterminal node of type "A" as its left hand side and a graph named "some-graph" as its right hand side.

- **name-of-lhs-node** – the label of the left hand side node. It has a numeric suffix which makes it unique, such as "A1", in case there are other rules for the left hand side's node type.

- **rhs-graph-spec** – made up of a graph name followed by any number of node labels or edge specifications. The node labels specify nodes which are in the graph, but which are not connected to any other node in the graph. (The edges which come into or out of any of their ports are all either input or output edges of the graph.) The edge specifications are in the form of a cons pair of port-specs. The first port-spec specifies the source port of the edge and the second specifies the sink port. A port-spec is a list of two elements – the first is a node name and the second is a port label (which is a number), specifying a particular port on the node. The port-spec may optionally contain a third element which is the string "input" or "output", telling the type of the port.

- **mapping** – an association list of port-specs such as those that appear in edge-lists, with the symbols "input" and "output" present as top-level sticky flags indicating the type of port. This specifies the mapping between the left and right hand sides. The first port-spec gives the left hand side node's port and the second port-spec gives the source (or sink) port of the edge to which the port is mapped.

- **st-thrus** – specifies which edges run right through the graph without being connected to any node. This is in the form of a list of cons pairs of port-specs in which the two port-specs of each pair specify the lhs node's input and output port, respectively, that are mapped to the straight-through edge.

- **node-type-constraints** – define the constraints on the type of the nodes in the rhs. This is an association list in which names of nodes in the rhs graph are associated with lambda expressions which place constraints on the type of nodes which can match the node they are associated with.

- **constraints** – these are forms to evaluate in order to check constraints on the graphs being matched with the rule's rhs. They are written in the constraint sublanguage described above.

- **att-transfer-specs** – association list in which attributes of the lhs node are associated with forms which when evaluated give values to be assigned to the attributes. The forms to be evaluated are expressed using the constraint sublanguage. In transformation rules, the node in the transformed graph which is to receive the attribute values can be specified by preceding the attribute and value by the name of the node. (If no node name is provided, the attribute values are defaultly assigned to the lhs node of the rule.)

- **doc** – documentation string used to construct a description of whatever has been recognized by the rule. The string may be followed by expressions written in the constraint sublanguage which can be used to fill in slots in the documentation string.

- **implementation** – implementation clues which give information about how a data structure is implemented and which are used to generate additional documentation.

- **transformation** – may be one of three values: NIL (the default value) means the rule is not to be used in transformations, i.e., it can only run forwards in the usual way; TRANSFORMATION-ONLY means the rule can only be used to expand the graph, i.e., it can only run backwards (in the Transform phase of the Recognizer which is separate from the Parsing phase); BOTH means the rule can be used in parsing as well as in transforming the graph, i.e., it can run forwards or backwards.

- **grammars** – list of names of grammars which are to contain this rule.

The Grammar Rules

Equality within an Epsilon

```
(defrule EWE>Equality-within-Epsilon EWE5
  (Equality-within-epsilon5
    ((minus5 3) . (Abs-Val5 1))
    ((Abs-Val5 2) . (lessp5 1)))
  (input
    ((EWE5 1) . (minus5 1))
    ((EWE5 2) . (minus5 2))
    ((EWE5 3) . (lessp5 2))
   output
    ((EWE5 4) . (lessp5 3)))
  :doc
  ("determines whether ~A and ~A differ by less than ~A."
   (input-name> (p1> (EWE5 1)))
   (input-name> (p1> (EWE5 2)))
   (input-name> (p1> (EWE5 3))))
  :grammars (the-grammar))
```

Absolute Value

```
(defrule Abs-Val>prim-abs Abs-Val32
  (prim-abs32 abs32)
  (input
    ((Abs-Val32 1) . (abs32 1))
   output
    ((Abs-Val32 2) . (abs32 2)))
  :doc
    ("computes the absolute value of ~A"
     (input-name> (p1> (Abs-Val32 1))))
  :grammars (the-grammar))
```

```
(defrule Abs-Val>Absolute-Value Abs-Val1
  (Absolute-Value1
    ((positive1 2) . (null-test1 1))
    negate1)
  (input
    ((Abs-Val1 1) . (negate1 1))
    ((Abs-Val1 1) . (positive1 1))
   output
    ((Abs-Val1 2) . (negate1 2)))
  :st-thrus
    (((Abs-Val1 1) . (Abs-Val1 2)))
  :constraints
    ((ce-equal (ce-from (p1> (negate1 2)) (p1> (Abs-Val1 2)))
               (success-ce (n> null-test1)))
     (ce-equal (ce-from (p1> (Abs-Val1 1)) (p1> (Abs-Val1 2)))
               (failure-ce (n> null-test1))))
  :doc
    ("computes the absolute value of ~A."
      (input-name> (p1> (Abs-Val1 1))))
  :grammars (the-grammar))
```

Sum of Squares

```
(defrule SOS>Sum-of-Squares SOS1
  (Sum-of-Squares1
    ((SQ1 2) . (plus1 2))
    ((SQ2 2) . (plus1 1)))
  (input
    ((SOS1 1) . (SQ1 1))
    ((SOS1 2) . (SQ2 1))
   output
    ((SOS1 3) . (plus1 3)))
  :doc
  ("computes the sum of the squares of ~A and ~A."
   (input-name> (p1> (SOS1 1)))
   (input-name> (p1> (SOS1 2))))
  :grammars (the-grammar))
```

Square

```
(defrule SQ>use-times SQ4
  (use-times2 times2)
  (input
    ((SQ4 1) . (times2 2))
    ((SQ4 1) . (times2 1))
   output
    ((SQ4 2) . (times2 3)))
  :doc
  ("computes the square of ~A" (input-name> (p1> (SQ4 1))))
  :grammars (the-grammar))

(defrule SQ>use-square SQ3
  (use-square3 square3)
  (input
    ((SQ3 1) . (square3 1))
   output
    ((SQ3 2) . (square3 2)))
  :doc
  ("computes the square of ~A." (input-name> (p1> (SQ3 1))))
  :grammars (the-grammar))
```

Average

```
(defrule Average>avg Average30
  (avg30
    ((plus30 3) . (halve30 1)))
  (input
    ((Average30 1) . (plus30 1))
    ((Average30 2) . (plus30 2))
   output
    ((Average30 3) . (halve30 2)))
  :doc
    ("computes the average of ~A and ~A"
     (input-name> (p1> (Average30 1)))
     (input-name> (p1> (Average30 2))))
  :grammars (the-grammar))
```

Generation

```
(defrule Gen>generation Gen2
  (generation2
    ((any-f2 2 output)))
  (input
    ((Gen2 1) . (any-f2 1)))
  :st-thrus
    (((Gen2 1) . (Gen2 2)))
  :node-type-constraints
    ((any-f2 . (lambda (nt) T)))
  :constraints
    ((feeds-back (p1> (any-f2 2)) (p1> (any-f2 1)))
     (in-a-loop (n> any-f2)))
  :att-transfer-specs
    ((function-info (function-info (n> any-f2)))
     (control-env (outside-ce (control-env (n> any-f2)))))
  :doc
    ("generates the elements of ~A by repeatedly applying ~
      ~&~A the function to the result of the preceding~
      ~&~A application of that function"
     (input-name> (p1> (Gen2 1)))
     (function-type (function-info (n> any-f2))))
  :grammars (the-grammar))
```

Map

```
(defrule Map>mapping Map3
  (mapping3 any-f3)
  (input
    ((Map3 1) . (any-f3 1))
   output
    ((Map3 2) . (any-f3 2)))
  :node-type-constraints
    ((any-f3 . (lambda (nt) T)))
  :constraints ((in-a-loop (n> any-f3)))
  :att-transfer-specs
    ((function-info (function-info (n> any-f3)))
     (control-env (outside-ce (control-env (n> any-f3)))))
  :doc
    ("applies the function ~A to each element of the input stream"
     (node-type (n> any-f3)))
  :grammars (the-grammar))
```

Test Predicate

```
(defrule Test-Predicate>pred-null Test-Predicate5
  (pred-null5
    ((predicate5 2) . (null-test5 1)))
  (input
    ((Test-Predicate5 1) . (predicate5 1)))
  :node-type-constraints
    ((predicate5 . (lambda (n) (predicate? n))))
  :att-transfer-specs
    ((predicate-info (function-info (n> predicate5)))
     (success-ce (failure-ce (n> null-test5)))
     (failure-ce (success-ce (n> null-test5)))
     (control-env (control-env (n> null-test5))))
  :doc
    ("tests ~A using the predicate ~A"
     (input-name> (p1> (Test-Predicate5 1)))
     (function-type (function-info (n> predicate5))))
  :grammars (the-grammar))
```

Filter

```
(defrule Filter>filtering Filter4
  (filtering4
    Test-Predicate4)
  (input
    ((Filter4 1) . (Test-Predicate4 1)))
  :st-thrus
    (((Filter4 1) . (Filter4 2)))
  :constraints
   ((loop for p in (p> (Filter4 2))
          for success-ce = (success-ce (nt-n> Test-Predicate4))
          unless (ce-le (ce-used-in p) success-ce)
             do (remove-output-mapping p '(Filter4 2))
                (remove-st-thru-input-mappings p '(Filter4 1))
           ;; the constraint holds if (Filter4 1) and
           ;; (Filter4 2) still have a mapping which means
           ;; for at least one p, the ce-le constraint held
           finally (return (and (p> (Filter4 1))
                                (p> (Filter4 2)))))
    (in-a-loop (nt-n> Test-Predicate4)))
  :att-transfer-specs
    ((predicate-info (predicate-info (nt-n> Test-Predicate4)))
     (control-env (outside-ce (control-env (nt-n> Test-Predicate4)))))
  :doc
    ("filters the elements of the input stream using the predicate ~A"
     (function-type (predicate-info (nt-n> Test-Predicate4))))
  :grammars (the-grammar))
```

Accumulation

```
(defrule Accum>accumulation Accum6
  (accumulation6
    ((any-f6 3 output)))
  (input
    ((Accum6 1) . (any-f6 1))
    ((Accum6 2) . (any-f6 2)))
  :st-thrus
    (((Accum6 2) . (Accum6 3)))
  :node-type-constraints
    ((any-f6 . (lambda (n) (binary-aggreg-function? n))))
  :constraints
    ((feeds-back (p1> (any-f6 3)) (p1> (any-f6 2))))
  :att-transfer-specs
    ((function-info (function-info (n> any-f6)))
     (control-env (outside-ce (control-env (n> any-f6)))))
  :doc
    ("accumulates the values of the input stream using the function ~A"
     (function-type (function-info (n> any-f6))))
 :grammars (the-grammar))
```

Truncate

```
(defrule Trunc>truncate Trunc7
  (truncate7
    Test-Predicate7)
  (input
    ((Trunc7 1) . (Test-Predicate7 1)))
  :st-thrus
    (((Trunc7 1) . (Trunc7 2)))
  :constraints
    ((loop for p in (p> (Trunc7 2))
           for failure-ce = (failure-ce (nt-n> Test-Predicate7))
           unless (ce-le (ce-used-in p) failure-ce)
             do (remove-output-mapping p '(Trunc7 2))
                (remove-st-thru-input-mappings p '(Trunc7 1))
           finally (return (and (p> (Trunc7 2))
                                (p> (Trunc7 1)))))
     (exit-predicate (nt-n> Test-Predicate7)))
  :att-transfer-specs
    ((predicate-info (predicate-info (nt-n> Test-Predicate7)))
     (termination-ce (success-ce (nt-n> Test-Predicate7)))
     (continuation-ce (failure-ce (nt-n> Test-Predicate7)))
     (control-env (outside-ce (control-env (nt-n> Test-Predicate7)))))
  :doc
    ("outputs the elements of the input stream up to but not including ~
      ~&the one that passes the predicate ~A"
     (function-type
       (predicate-info (nt-n> Test-Predicate7))))
  :grammars (the-grammar))
```

Truncate Inclusive

```
(defrule Trunc-Inc>truncate-inclusive Trunc-Inc8
  (truncate-inclusive8
    Test-Predicate8)
  (input
    ((Trunc-Inc8 1) . (Test-Predicate8 1)))
  :st-thrus
    (((Trunc-Inc8 1) . (Trunc-Inc8 2)))
  :constraints
  ((loop for p in (p> (Trunc-Inc8 2))
         unless (ce-le (control-env (nt-n> Test-Predicate8))
                       (ce-used-in p))
           do (remove-output-mapping p '(Trunc-Inc8 2))
              (remove-st-thru-mapping p '(Trunc-Inc8 1))
         finally (return (and (p> (Trunc-Inc8 1))
                              (p> (Trunc-Inc8 2)))))
     (exit-predicate (nt-n> Test-Predicate8)))
  :att-transfer-specs
    ((predicate-info (predicate-info (nt-n> Test-Predicate8)))
     (termination-ce (success-ce (nt-n> Test-Predicate8)))
     (continuation-ce (failure-ce (nt-n> Test-Predicate8)))
     (control-env (outside-ce (control-env (nt-n> Test-Predicate8)))))
  :doc
    ("outputs the elements of the input stream up to and including ~
      ~&the first one that passes the predicate ~A"
      (function-type (predicate-info (nt-n> Test-Predicate8))))
  :grammars (the-grammer))
```

Co-Truncate

```
(defrule Co-Trunc>co-truncate Co-Trunc9
  (co-truncate9
    Test-Predicate9)
  (input
    ((Co-Trunc9 1) . (Test-Predicate9 1)))
  :st-thrus
    (((Co-Trunc9 2) . (Co-Trunc9 3)))
  :constraints
   ((loop for p in (p> (Co-Trunc9 3))
           for failure-ce = (failure-ce (nt-n> Test-Predicate9))
           unless (ce-le (ce-used-in p) failure-ce)
             do (remove-output-mapping p '(Co-Trunc9 3))
                (remove-st-thru-input-mappings p '(Co-Trunc9 2))
           finally (return (and (p> (Co-Trunc9 3))
                                (p> (Co-Trunc9 2)))))
     (exit-predicate (nt-n> Test-Predicate9)))
  :att-transfer-specs
    ((predicate-info (predicate-info (nt-n> Test-Predicate9)))
     (termination-ce (success-ce (nt-n> Test-Predicate9)))
     (continuation-ce (failure-ce (nt-n> Test-Predicate9)))
     (control-env (outside-ce (control-env (nt-n> Test-Predicate9)))))
  :doc
    ("outputs the elements of the second input stream up to but not ~
      ~&including the one corresponding to the element of the first ~
      ~&input stream that passes the predicate ~A"
      (function-type (predicate-info (nt-n> Test-Predicate9))))
 :grammars (the-grammar))
```

Co-Truncate Inclusive

```
(defrule Co-Trunc-Inc>co-truncate-inclusive Co-Trunc-Inc10
  (co-truncate-inclusive10
    Test-Predicate10)
  (input
    ((Co-Trunc-Inc10 1) . (Test-Predicate10 1)))
  :st-thrus
    (((Co-Trunc-Inc10 2) . (Co-Trunc-Inc10 3)))
  :constraints
    ((loop for p in (p> (Co-Trunc-Inc10 3))
         unless (ce-le (control-env (nt-n> Test-Predicate10))
                       (ce-used-in p))
           do (remove-output-mapping p '(Co-Trunc-Inc10 3))
              (remove-st-thru-mapping p '(Co-Trunc-Inc10 2))
         finally (return (and (p> (Co-Trunc-Inc10 2))
                              (p> (Co-Trunc-Inc10 3)))))
     (exit-predicate (nt-n> Test-Predicate10)))
  :att-transfer-specs
    ((predicate-info (predicate-info (nt-n> Test-Predicate10)))
     (termination-ce (success-ce (nt-n> Test-Predicate10)))
     (continuation-ce (failure-ce (nt-n> Test-Predicate10)))
     (control-env (outside-ce (control-env (nt-n> Test-Predicate10)))))
  :doc
    ("outputs the elements of the second input stream up to and ~
      ~&including the one corresponding to the first element of the ~
      ~&first input stream that passes the predicate ~A"
      (function-type (predicate-info (nt-n> Test-Predicate10))))
 :grammars (the-grammar))
```

Earliest

```
(defrule Earl>earliest Earl11
  (earliest11
    Test-Predicate11)
  (input
    ((Earl11 1) . (Test-Predicate11 1)))
  :st-thrus
    (((Earl11 1) . (Earl11 2)))
  :constraints
    ((loop for p in (p> (Earl11 2))
           for outside-ce =
                (outside-ce (control-env (nt-n> Test-Predicate11)))
           unless (ce-le (ce-used-in p) outside-ce)
             do (remove-output-mapping p '(Earl11 2))
                (remove-st-thru-input-mappings p '(Earl11 1))
           finally (return (and (p> (Earl11 2))
                                (p> (Earl11 1)))))
     (exit-predicate (nt-n> Test-Predicate11)))
  :att-transfer-specs
    ((predicate-info (predicate-info (nt-n> Test-Predicate11)))
     (termination-ce (success-ce (nt-n> Test-Predicate11)))
     (continuation-ce (failure-ce (nt-n> Test-Predicate11)))
     (control-env (outside-ce (control-env (nt-n> Test-Predicate11)))))
  :doc
    ("outputs the first element of the input sequence which passes the ~
      ~&predicate ~A"
       (function-type (predicate-info (nt-n> Test-Predicate11))))
  :grammars (the-grammar))
```

Co-Earliest

```
(defrule Co-Earl>co-earliest Co-Earl12
  (co-earliest12
    Test-Predicate12)
  (input
    ((Co-Earl12 1) . (Test-Predicate12 1)))
  :st-thrus
    (((Co-Earl12 2) . (Co-Earl12 3)))
  :constraints
    ((loop for p in (p> (Co-Earl12 3))
           for outside-ce =
                (outside-ce (control-env (nt-n> Test-Predicate12)))
           unless (ce-le (ce-used-in p) outside-ce)
             do (remove-output-mapping p '(Co-Earl12 3))
                (remove-st-thru-input-mappings p '(Co-Earl12 2))
           finally (return (and (p> (Co-Earl12 3))
                                (p> (Co-Earl12 2)))))
     (exit-predicate (nt-n> Test-Predicate12)))
  :att-transfer-specs
    ((predicate-info (predicate-info (nt-n> Test-Predicate12)))
     (termination-ce (success-ce (nt-n> Test-Predicate12)))
     (continuation-ce (failure-ce (nt-n> Test-Predicate12)))
     (control-env (outside-ce (control-env (nt-n> Test-Predicate12)))))
  :doc
    ("outputs the first element of the input sequence which passes the ~
      ~&predicate ~A"
      (function-type (predicate-info (nt-n> Test-Predicate12))))
  :grammars (the-grammar))
```

SubList Enumeration

```
(defrule SLE>sublist-enum SLE14
  (sublist-enum14
    ((Gen14 2) . (Trunc14 1)))
  (input
    ((SLE14 1) . (Gen14 1))
   output
    ((SLE14 2) . (Trunc14 2)))
  :constraints
    ((co-occur (nt-n> Gen14) (nt-n> Trunc14))
     (eq (function-type (function-info (nt-n> Gen14)))
       'cdr)
     (eq (function-type (predicate-info (nt-n> Trunc14)))
       'null))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Gen14))))
  :doc
    ("enumerates the successive sublists of ~A"
     (input-name> (p1> (SLE14 1))))
  :grammars (the-grammar))
```

List Enumeration

```
(defrule LE>list-enum LE15
  (list-enum15 ((SLE15 2) . (Map15 1)))
  (input
    ((LE15 1) . (SLE15 1))
   output
    ((LE15 2) . (Map15 2)))
  :constraints
    ((co-occur (nt-n> SLE15) (nt-n> Map15))
     (eq (function-type (function-info (nt-n> Map15))) 'car))
  :att-transfer-specs
    ((control-env (control-env (nt-n> SLE15))))
  :doc
    ("enumerates the elements of ~A" (input-name> (p1> (LE15 1))))
  :grammars (the-grammar))
```

List Reverse

The constant input of NIL as the init of the Accumulation is incorporated into the Accum node, yielding an Accum-from-constant node in the List-Reverse grammar rule. The rule constrains the constant to be NIL. We use the transformation rule for Trans-Accum to transform an instance of the Accumulation cliche in which the initial value is a constant to an Accum-from-constant node.

```
(defrule List-Reverse>revlist List-Reverse18
  (revlist18
    ((LE18 2) . (Accum-from-constant18 1)))
  (input
    ((List-Reverse18 1) . (LE18 1))
   output
    ((List-Reverse18 2) . (Accum-from-constant18 2)))
  :constraints
    ((co-occur (nt-n> LE18) (n> Accum-from-constant18))
     (eq (init-value (n> Accum-from-constant18))
         'NIL-source)
     (eq (function-type (accum-function-info
                          (n> Accum-from-constant18)))
         'cons)
     ;; constraint for the cliche Last:
     (loop for p-out in (p> (Accum-from-constant18 2))
           unless (ce-le (ce-used-in p-out)
                  (control-env (n> Accum-from-constant18)))
             do (remove-output-mapping
                  p-out '(Accum-from-constant18 2))
           finally (return (p> (Accum-from-constant18 2)))))
  :att-transfer-specs
    ((control-env (control-env (nt-n> LE18))))
  :doc
    ("computes the reverse of the list ~A"
     (input-name> (p1> (List-Reverse18 1))))
  :grammars (the-grammar))
```

List Length

The constant input of 0 to Count is incorporated into the Count, yielding another source which is incorporated into the Co-Earliest. Therefore, List-Length uses a node called Co-E-of-Count-from-Zero which means a Co-Earliest of a Count starting with 0. This requires the use of the Trans-Co-E-Count transformation rule.

```
(defrule List-Length>ll List-Length22
  (ll22 ((Gen22 2) . (Co-E-of-Count-from-Zero22 1)))
  (input
    ((List-Length22 1) . (Gen22 1))
   output
    ((List-Length22 2) . (Co-E-of-Count-from-Zero22 2)))
  :constraints
    ((eq (function-type (function-info (nt-n> Gen22))) 'cdr)
     (eq (function-type
           (co-earl-predicate-info (n> Co-E-of-Count-from-Zero22)))
         'null)
     (co-occur (nt-n> Gen22) (n> Co-E-of-Count-from-Zero22)))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Gen22))))
  :doc
    ("computes the length of ~A" (input-name> (p1> (List-Length22 1))))
  :grammars (the-grammar))
```

Count

```
(defrule Count>cnt Count19
  (cnt19 Gen19)
  (input  ((Count19 1) . (Gen19 1))
   output ((Count19 2) . (Gen19 2)))
  :constraints
    ((eq (function-type (function (nt-n> Gen19))) 'one-plus))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Gen19)))
     (init-value (source-type (p1> (Count19 1)))))
  :doc
    ("generates a series of numbers starting with ~A"
      (input-name> (p1> (Count19 1)))))
```

Positive Sublist

```
(defrule Pos-Sublist>positive-sublist Pos-Sublist23
  (positive-sublist23
    ((LE23 2) . (Filter23 1))
    ((Filter23 2) . (Accum-from-constant23 1)))
  (input
    ((Pos-Sublist23 1) . (LE23 1))
   output
    ((Pos-Sublist23 2) . (Accum-from-constant23 2)))
  :constraints
    ((co-occur (nt-n> LE23) (nt-n> Filter23))
     (co-occur (n> Accum-from-constant23) (nt-n> Filter23))
     (eq (function-type (accum-function-info
                          (n> Accum-from-constant23)))
         'cons)
     (eq (init-value (n> Accum-from-constant23)) 'NIL-source)
     (eq (function-type (predicate-info (nt-n> Filter23)))
         'positive))
  :att-transfer-specs
    ((control-env (control-env (nt-n> LE23))))
  :doc
    ("returns a list of the positive elements of ~A"
      (input-name> (p1> (Pos-Sublist23 1))))
  :grammars (the-grammar))
```

Sequential List Search

```
(defrule Seq-List-Search>list-search Seq-List-Search24
  (list-search24
    ((LE24 2) . (Earl24 1)))
  (input
    ((Seq-List-Search24 1) . (LE24 1))
   output
    ((Seq-List-Search24 2) . (Earl24 2)))
  :constraints
    ((co-occur (nt-n> LE24) (nt-n> Earl24)))
  :att-transfer-specs
    ((control-env (control-env (nt-n> LE24)))
     (success-ce (termination-ce (nt-n> Earl24)))
     (failure-ce (termination-ce (nt-n> LE24)))
     (predicate-info (predicate-info (nt-n> Earl24))))
  :doc
    ("sequentially searches the elements of ~A ~
      ~&for one that satisfies the test ~A"
     (input-name> (p1> (Seq-List-Search24 1)))
     (function-type (predicate-info (nt-n> Earl24))))
  :grammars (the-grammar))
```

Member

```
(defrule Member>lisp-member Member27
  (lisp-member27
    ((SLE27 2) . (Map27 1))
    ((Map27 2) . (Co-Earl27 1))
    ((SLE27 2) . (Co-Earl27 2))
    ignore27)
  (input
    ((Member27 1) . (ignore27 1))
    ((Member27 2) . (SLE27 1))
   output
    ((Member27 3) . (Co-Earl27 3)))
  :constraints
    ((co-occur (nt-n> SLE27) (nt-n> Map27))
     (co-occur (nt-n> SLE27) (nt-n> Co-Earl27))
     (eq (function-type (function-info (nt-n> Map27))) 'car)
     (eq (function-type (predicate-info (nt-n> Co-Earl27))) 'eq)
     (partially-evaluated (predicate-info (nt-n> Co-Earl27)))
     (eq (closed-wrt (predicate-info (nt-n> Co-Earl27)) 1)
         (p1> (Member27 1))))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Map27)))
     (success-ce (termination-ce (nt-n> Co-Earl27)))
     (failure-ce (termination-ce (nt-n> SLE27))))
  :doc
    ("uses EQ to determine if ~A is a member of the list ~A"
      (input-name> (p1> (Member27 1)))
      (input-name> (p1> (Member27 2))))
  :grammars (the-grammar))
```

Set Member

```
(defrule Set-Membership>ht-mem Set-Membership33
  (ht-mem33 Hash-Table-Member33)
  (input
    ((Set-Membership33 1) . (Hash-Table-Member33 1))
    ((Set-Membership33 2) . (Hash-Table-Member33 2)))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Hash-Table-Member33)))
     (success-ce (success-ce (nt-n> Hash-Table-Member33)))
     (failure-ce (failure-ce (nt-n> Hash-Table-Member33))))
  :doc
    ("determines whether or not ~A is an element of the set ~A."
     (input-name> (p1> (Set-Membership33 2)))
     (input-name> (p1> (Set-Membership33 1))))
  :implementation
    (("~&The set is implemented as a Hash Table."))
  :grammars (the-grammar))
```

Hash Table Member

```
(defrule Hash-Table-Member>hash-fetch-mem Hash-Table-Member34
  (hash-fetch-mem34
    ((hash34 2) . (Fetch-Bucket34 2))
    ((Fetch-Bucket34 3) . (Bucket-Member34 1)))
  (input
    ((Hash-Table-Member34 1) . (Fetch-Bucket34 1))
    ((Hash-Table-Member34 2) . (hash34 1))
    ((Hash-Table-Member34 2) . (Bucket-Member34 2)))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Bucket-Member34)))
     (success-ce (success-ce (nt-n> Bucket-Member34)))
     (failure-ce (failure-ce (nt-n> Bucket-Member34))))
  :doc
    ("determines whether or not ~A is an element of the Hash Table ~A."
     (input-name> (p1> (Hash-Table-Member34 2)))
     (input-name> (p1> (Hash-Table-Member34 1))))
  :grammars (the-grammar))
```

Fetch Bucket

```
(defrule Fetch-Bucket>lookup-index Fetch-Bucket35
  (lookup-index35 aref35)
  (input
    ((Fetch-Bucket35 1) . (aref35 1))
    ((Fetch-Bucket35 2) . (aref35 2))
   output
    ((Fetch-Bucket35 3) . (aref35 3)))
  :att-transfer-specs
    ((function-info (function-info (n> aref35)))
     (control-env (control-env (n> aref35))))
  :doc
    ("looks up a bucket in ~A using ~A as an index"
     (input-name> (p1> (Fetch-Bucket35 1)))
     (input-name> (p1> (Fetch-Bucket35 2))))
  :implementation
    (("~&The Hash Table is implemented as an Array of ~
       ~&buckets, indexed by hash-code."))
  :grammars (the-grammar))
```

Bucket Member

```
(defrule Bucket-Member>ord-list-mem Bucket-Member36
  (ord-list-mem36
    Ordered-List-Member36)
  (input
    ((Bucket-Member36 1) . (Ordered-List-Member36 1))
    ((Bucket-Member36 2) . (Ordered-List-Member36 2)))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Ordered-List-Member36)))
     (success-ce (success-ce (nt-n> Ordered-List-Member36)))
     (failure-ce (failure-ce (nt-n> Ordered-List-Member36))))
  :doc
    ("An Ordered-List-Member is used to determine whether or not ~
      ~&~A is in the fetched bucket, ~A"
     (input-name> (p1> (Bucket-Member36 2)))
     (input-name> (p1> (Bucket-Member36 1))))
  :implementation
    (("~&The buckets are implemented as Ordered Lists.")
     ((if (eq (function-type
                (predicate-info (nt-n> Ordered-List-Member36)))
              'string-gt)
          "~&They are ordered lexicographically."))
     ("~&The elements in the buckets are ~A."
       (domain-of-f
         (function-type
           (predicate-info (nt-n> Ordered-List-Member36))))))
  :grammars (the-grammar))
```

Ordered List Member

```
(defrule Ordered-List-Member>le-trunc-earl Ordered-List-Member37
  (le-trunc-earl37
    ((LE37 2) . (Trunc37 1))
    ((Trunc37 2) . (Earl37 1))
    ignore37
    ignore38)
  (input
    ((Ordered-List-Member37 1) . (LE37 1))
    ((Ordered-List-Member37 2) . (ignore37 1))
    ((Ordered-List-Member37 2) . (ignore38 1)))
  :constraints
    ((co-occur (nt-n> LE37) (nt-n> Trunc37))
     (co-occur (nt-n> LE37) (nt-n> Earl37))
     (partially-evaluated (predicate-info (nt-n> Trunc37)))
     (eq (closed-wrt (predicate-info (nt-n> Trunc37)) 2)
         (p1> (Ordered-List-Member37 2)))
     (eq (function-type (predicate-info (nt-n> Earl27))) 'equal)
     (partially-evaluated (predicate-info (nt-n> Earl37)))
     (eq (closed-wrt (predicate-info (nt-n> Earl37)) 2)
         (p1> (Ordered-List-Member37 2))))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Trunc37)))
     (predicate-info (predicate-info (nt-n> Trunc37)))
     (success-ce (termination-ce (nt-n> Earl37)))
     (failure-ce (ce-sum
                  (termination-ce (nt-n> LE37))
                  (termination-ce (nt-n> Trunc37)))))
  :doc
    ("determines whether or not ~A is an element of the ordered list ~A"
     (input-name> (p1> (Ordered-List-Member37 2)))
     (input-name> (p1> (Ordered-List-Member37 1))))
  :grammars (the-grammar))
```

Sum

```
(defrule Sum>accum-from-zero Sum39
  (accum-from-zero39 Accum-from-constant39)
  (input
    ((Sum39 1) . (Accum-from-constant39 1))
   output
    ((Sum39 2) . (Accum-from-constant39 2)))
  :constraints
    ((eq (init-value (n> Accum-from-constant39)) 0)
     (eq (function-type (accum-function-info (n> Accum-from-constant39)))
         'plus))
  :att-transfer-specs
    ((control-env (control-env (n> Accum-from-constant39))))
  :doc
    ("accumulates the sum of the successive values of ~A"
     (input-name> (p1> (Sum39 1))))
  :grammars (the-grammar))
```

Sum Elements

```
(defrule Sum-Elements>sum-elems Sum-Elements40
  (sum-elems40 ((LE40 2) . (Sum40 1)))
  (input  ((Sum-Elements40 1) . (LE40 1))
   output ((Sum-Elements40 2) . (Sum40 2)))
  :constraints
    ((co-occur (nt-n> LE40) (nt-n> Sum40))
     ;; constraint for Last:
     (loop for p-out in (p> (Sum-Elements40 2))
           unless (ce-le (ce-used-in p-out) (control-env (nt-n> Sum40)))
             do (remove-output-mapping p-out '(Sum-Elements40 2))
           finally (return (p> (Sum-Elements40 2)))))
  :att-transfer-specs
    ((control-env (control-env (nt-n> LE40))))
  :doc
    ("computes the sum of the elements of the list ~A"
     (input-name> (p1> (Sum-Elements40 1))))
  :grammars (the-grammar))
```

List Average

```
(defrule List-Average>list-avg List-Average41
  (list-avg41
    ((Sum-Elements41 2) . (divide41 1))
    ((List-Length41 2) . (divide41 2)))
  (input
    ((List-Average41 1) . (Sum-Elements41 1))
    ((List-Average41 1) . (List-Length41 1))
   output
    ((List-Average41 2) . (divide41 3)))
  :att-transfer-specs
    ((control-env (control-env (n> divide41))))
  :doc
    ("computes the average of the elements in the list "A"
     (input-name> (p1> (List-Average41 1))))
  :grammars (the-grammar))
```

Vector Enumeration

Vector Enumeration of elements I to N, not including the Nth element.

```
(defrule VE-I-to-N>vec-enum-i-n VE-I-to-N
  (vec-enum-i-n42
    ((Count42 2) . (Trunc42 1))
    ((Trunc42 2) . (Map42 1))
    ignore42 ignore43)
  (input
    ((VE-I-to-N42 1) . (ignore42 1))
    ((VE-I-to-N42 2) . (Count42 1))
    ((VE-I-to-N42 3) . (ignore43 1))
   output
    ((VE-I-to-N42 4) . (Map42 2)))
  :constraints
    ((eq (function-type (predicate-info (nt-n> Trunc42))) 'gte)
     (partially-evaluated (predicate-info (nt-n> Trunc42)))
     (eq (closed-wrt (predicate-info (nt-n> Trunc42)) 2)
         (p1> (VE-I-to-N42 2)))
     (eq (function-type (function-info (nt-n> Map42))) 'aref)
     (partially-evaluated (function-info (nt-n> Map42)))
     (eq (closed-wrt (function-info (nt-n> Map42)) 1)
         (p1> (Map42 1))))
     (co-occur (nt-n> Map42) (nt-n> Trunc42))
     (co-occur (nt-n> Map42) (nt-n> Count42)))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Map42))))
  :doc
    ("enumerates the elements of the vector ~A from I = ~A up to N= ~A"
     (input-name> (p1> (VE-I-to-N42 1)))
     (input-name> (p1> (VE-I-to-N42 2)))
     (input-name> (p1> (VE-I-to-N42 3))))
  :implementation
    (("~&The vector ~A is implemented as an Array."
      (input-name> (p1> (VE-I-to-N42 1)))))
  :grammars (the-grammar))
```

Vector Enumeration of the first N elements of a vector

```
(defrule VE-first-N>VE-constant-N VE-first45
  (VE-constant-N45
    VE-constant-to-N45)
  (input
    ((VE-first-N45 1) . (VE-constant-to-N45 1))
    ((VE-first-N45 2) . (VE-constant-to-N45 2))
   output
    ((VE-first-N45 3) . (VE-constant-to-N45 3)))
  :constraints
    ((eq (init-value (n> VE-constant-to-N45)) 0))
  :att-transfer-specs
    ((control-env (control-env (n> VE-constant-to-N45))))
  :doc
    ("enumerates the first N = ~A of the vector ~A"
     (input-name> (p1> (VE-first45 2)))
     (input-name> (p1> (VE-first45 1))))
  :grammars (the-grammar))
```

Transformation Rules

A transformation rule is specified by a pair of rules, one of which runs forward and the other of which runs backward. The "forward-running" rule serves to find subgraphs in the input graph which may be transformed. They replace the subgraph temporarily with an intermediate node. This intermediate node is then expanded into the subgraph which is the result of the transformation using a "backward-running" rule. For example, the forward-running rule "Trans-Accum>find-accum" reduces to an intermediate node of type Trans-Accum. This node is then expanded using the "backward-running" rule "Trans-accum>accum-constant". The TRANSFORMATION option in the rule definition specifies whether a rule is to be run forward or backward. A value of "NIL" means the rule can only run forward, which is its normal direction in parsing. A value of "TRANSFORMATION-ONLY" means the rule can only run backward to expand intermediate nodes. A value of "BOTH" means the rule can be run in either direction.

Attributes must be transferred between the graphs involved in the transformation. This is done by transferring attributes from the rhs of the "forward-running" rule to its lhs (as usual) and then when the expansion is done, transferring these attributes to the expanded graph. In order for the attributes to be assigned to the expanded graph, the nodes to which they should be assigned must be specified. This is done in the attribute transfer specifications by preceding the attribute and value in the specification by the name of the node which is to receive the attribute.

The following pair of rules transform and instance of an Accumulation which always receives a constant as its initial value (i.e., its second input) to an instance of Accum-from-constant.

```
(defrule Trans-Accum>find-accum Trans-Accum17
  (find-accum17
    Accum17)
  (input
    ((Trans-Accum17 1) . (Accum17 1))
    ((Trans-Accum17 2) . (Accum17 2))
   output
    ((Trans-Accum17 3) . (Accum17 3)))
  :constraints
    ((any-source (p1> (Trans-Accum17 2))))
  :att-transfer-specs
    ((accum-function-info (function-info (nt-n> Accum17)))
     (init-value (source-type (p1> (Trans-Accum17 2))))
     (control-env (control-env (nt-n> Accum17))))
  :transformation NIL
  :grammars (the-grammar))

(defrule Trans-Accum>accum-constant Trans-Accum16
  (accum-constant16
    Accum-from-constant16
    ignore16)
  (input
    ((Trans-Accum16 1) . (Accum-from-constant16 1))
    ((Trans-Accum16 2) . (ignore16 1))
   output
    ((Trans-Accum16 3) . (Accum-from-constant16 2)))
  :att-transfer-specs
    ((Accum-from-constant16 accum-function-info
                            (accum-function-info (nt-n> Trans-Accum16)))
     (Accum-from-constant16 init-value (init-value (nt-n> Trans-Accum16)))
     (Accum-from-constant16 control-env (control-env (nt-n> Trans-Accum16))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

The following pair of rules specify a transformation of a node of input arity 2 (output arity 1) to one of input arity 1 as long as the first input is a loop constant.

```
(defrule Trans-to-Arity-One-Ig-First>find-l-c-to-first
  Trans-to-Arity-One-Ig-First26
  (find-l-c-to-first26
   any-f26)
  (input
    ((Trans-to-Arity-One-Ig-First26 1) . (any-f26 1))
    ((Trans-to-Arity-One-Ig-First26 2) . (any-f26 2))
   output
    ((Trans-to-Arity-One-Ig-First26 3) . (any-f26 3)))
  :node-type-constraints
    ((any-f26 . (lambda (n) T)))
  :constraints
    ((in-a-loop (n> any-f26))
     (loop-constant (p1> (any-f26 1))))
  :att-transfer-specs
    ((control-env (control-env (n> any-f26)))
     (function-info
       (create-function-info
         (function-type (function-info (n> any-f26)))
         (create-closed-wrt-alist
           1 (p1> (Trans-to-Arity-One-Ig-First26 1))))))
  :transformation NIL
  :grammars (the-grammar))
```

```
(defrule Trans-to-Arity-One-Ig-First>one-input
  Trans-to-Arity-One-Ig-First25
  (one-input25
   one-input-f25
   ignore25)
  (input
    ((Trans-to-Arity-One-Ig-First25 1) . (ignore25 1))
    ((Trans-to-Arity-One-Ig-First25 2) . (one-input-f25 1))
   output
    ((Trans-to-Arity-One-Ig-First25 3) . (one-input-f25 2)))
  :att-transfer-specs
    ((one-input-f25 function-info
                    (function-info
                      (nt-n> Trans-to-Arity-One-Ig-First25)))
     (one-input-f25 control-env
                    (control-env
                      (nt-n> Trans-to-Arity-One-Ig-First25))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

The following pair of rules specify a transformation of a node of input arity 2 (output arity 1) to one of input arity 1 as long as the second input is a loop constant.

```
(defrule Trans-to-Arity-One-Ig-Second>find-l-c-to-second
  Trans-to-Arity-One-Ig-Second29
  (find-l-c-to-second29
   any-f29)
  (input
    ((Trans-to-Arity-One-Ig-Second29 1) . (any-f29 1))
    ((Trans-to-Arity-One-Ig-Second29 2) . (any-f29 2))
   output
    ((Trans-to-Arity-One-Ig-Second29 3) . (any-f29 3)))
  :node-type-constraints
    ((any-f29 . (lambda (n) T)))
  :constraints
    ((in-a-loop (n> any-f29))
     (loop-constant (p1> (any-f29 2))))
  :att-transfer-specs
    ((control-env (control-env (n> any-f29)))
     (function-info
       (create-function-info
         (function-type (function-info (n> any-f29)))
         (create-closed-wrt-alist
           2 (p1> (Trans-to-Arity-One-Ig-Second29 2))))))
  :transformation NIL
  :grammars (the-grammar))
```

```
(defrule Trans-to-Arity-One-Ig-Second>one-input
  Trans-to-Arity-One-Ig-Second28
  (one-input28
   ignore28
   one-input-f28)
  (input
    ((Trans-to-Arity-One-Ig-Second28 1) . (one-input-f28 1))
    ((Trans-to-Arity-One-Ig-Second28 2) . (ignore28 1))
   output
    ((Trans-to-Arity-One-Ig-Second28 3) . (one-input-f28 2)))
  :att-transfer-specs
    ((one-input-f28 function-info
                    (function-info
                      (nt-n> Trans-to-Arity-One-Ig-Second28)))
     (one-input-f28 control-env
                    (control-env
                      (nt-n> Trans-to-Arity-One-Ig-Second28))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

The next pair of rules transform an instance of a Count, whose starting number (first input) is 0 and which is being given as the second input to a Co-Earliest, into a single node which represents a Co-Earliest being given a constant source (i.e. Count from Zero) as its second input.

```
(defrule Trans-Co-E-Count>find-co-e-of-cnt Trans-Co-E-Count21
  (find-co-e-of-cnt21
    ((Count21 2) . (Co-Earl21 2)))
  (input
    ((Trans-Co-E-Count21 1) . (Co-Earl21 1))
    ((Trans-Co-E-Count21 2) . (Count21 1))
   output
    ((Trans-Co-E-Count21 3) . (Co-Earl21 3)))
  :constraints
    ((co-occur (nt-n> Count21) (nt-n> Co-Earl21))
     (eq (init-value (nt-n> Count21)) 0))
  :att-transfer-specs
    ((control-env (control-env (nt-n> Co-Earl21)))
     (co-earl-predicate-info (predicate-info (nt-n> Co-Earl21))))
  :transformation NIL
  :grammars (the-grammar))

(defrule Trans-Co-E-Count>co-e-of-cnt Trans-Co-E-Count20
  (co-e-of-cnt20 Co-E-of-Count-from-Zero20 ignore20)
  (input
    ((Trans-Co-E-Count20 1) . (Co-E-of-Count-from-Zero20 1))
    ((Trans-Co-E-Count20 2) . (ignore20 1))
   output
    ((Trans-Co-E-Count20 3) . (Co-E-of-Count-from-Zero20 2)))
  :att-transfer-specs
    ((Co-E-of-Count-from-Zero20
       co-earl-predicate-info
       (co-earl-predicate-info (nt-n> Trans-Co-E-Count20)))
     (Co-E-of-Count-from-Zero20
       control-env (control-env (nt-n> Trans-Co-E-Count20))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

The following pair of rules transform an instance of a Vector Enumeration of elements I up to N where I is a constant to an instance of a Vector Enumeration of N whose initial value is the constant.

```
(defrule Trans-VE-I-to-N>find-VE-I-to-N Trans-VE-I-to-N43
  (find-VE-I-to-N43
    VE-I-to-N43)
  (input
    ((Trans-VE-I-to-N43 1) . (VE-I-to-N43 1))
    ((Trans-VE-I-to-N43 2) . (VE-I-to-N43 2))
    ((Trans-VE-I-to-N43 3) . (VE-I-to-N43 3))
   output
    ((Trans-VE-I-to-N43 4) . (VE-I-to-N43 4)))
  :constraints
    ((any-source (p1> (VE-I-to-N43 3))))
  :att-transfer-specs
    ((init-value (source-type (p1> (VE-I-to-N43 3))))
     (control-env (control-env (nt-n> VE-I-to-N43))))
  :transformation NIL
  :grammars (the-grammar))

(defrule Trans-VE-I-to-N>VE-constant Trans-VE-I-to-N44
  (VE-constant44
    VE-constant-to-N44
    ignore44)
  (input
    ((Trans-VE-I-to-N44 1) . (VE-constant-to-N44 1))
    ((Trans-VE-I-to-N44 2) . (VE-constant-to-N44 2))
    ((Trans-VE-I-to-N44 3) . (ignore44 1))
   output
    ((Trans-VE-I-to-N44 4) . (VE-constant-to-N44 3)))
  :att-transfer-specs
    ((VE-constant-to-N44 init-value (init-value (n> Trans-VE-I-to-N44)))
     (VE-constant-to-N44 control-env (control-env (n> Trans-VE-I-to-N44))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

The following pair of rules transform an instance of a division by 2 into a "halve" operation.

```
(defrule Trans-Halve>find-halve Trans-Halve44
  (find-halve44
    divide44)
  (input
    ((Trans-Halve44 1) . (divide44 1))
    ((Trans-Halve44 2) . (divide44 2))
   output
    ((Trans-Halve44 3) . (divide44 3)))
  :constraints
    ((source? (p1> (Trans-Halve44 2)) 2))
  :att-transfer-specs
    ((control-env (control-env (n> divide44))))
  :transformation NIL
  :grammars (the-grammar))

(defrule Trans-Halve>halve-ignore-constant Trans-Halve45
  (halve-ignore-constant45
    halve45
    ignore45)
  (input
    ((Trans-Halve45 1) . (halve45 1))
    ((Trans-Halve45 2) . (ignore45 1))
   output
    ((Trans-Halve45 3) . (halve45 2)))
  :att-transfer-specs
    ((halve45 control-env (control-env (n> Trans-Halve45))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

These rules transform (GT x 0) to (POSITIVE x).

```
(defrule Trans-GT-to-Positive>find-gt
  Trans-GT-to-Positive20
  (find-gt20
    gt20)
  (input
    ((Trans-GT-to-Positive20 1) . (gt20 1))
    ((Trans-GT-to-Positive20 2) . (gt20 2))
   output
    ((Trans-GT-to-Positive20 3) . (gt20 3)))
  :constraints
    ((source? (p1> (Trans-GT-to-Positive20 2)) 0))
  :att-transfer-specs
    ((control-env (control-env (n> gt20))))
  :transformation NIL
  :grammars (the-grammar))

(defrule Trans-GT-to-Positive>positive-graph
  Trans-GT-to-Positive21
  (positive-graph21
    positive21
    ignore21)
  (input
    ((Trans-GT-to-Positive21 1) . (positive21 1))
    ((Trans-GT-to-Positive21 2) . (ignore21 1))
   output
    ((Trans-GT-to-Positive21 3) . (positive21 2)))
  :att-transfer-specs
    ((positive21 control-env
                 (control-env (nt-n> Trans-GT-to-Positive21))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

These rules transform (Plus x 1) to (One-Plus x).

```
(defrule Trans-Plus-to-One-Plus>find-plus
  Trans-Plus-to-One-Plus20
  (find-plus20
    plus20)
  (input
    ((Trans-Plus-to-One-Plus20 1) . (plus20 1))
    ((Trans-Plus-to-One-Plus20 2) . (plus20 2))
   output
    ((Trans-Plus-to-One-Plus20 3) . (plus20 3)))
  :constraints
    ((source? (p1> (Trans-Plus-to-One-Plus20 2)) 1))
  :att-transfer-specs
    ((control-env (control-env (n> plus20))))
  :transformation NIL
  :grammars (the-grammar))

(defrule Trans-Plus-to-One-Plus>one-plus-graph
  Trans-Plus-to-One-Plus21
  (one-plus-graph21 one-plus21 ignore21)
  (input
    ((Trans-Plus-to-One-Plus21 1) . (one-plus21 1))
    ((Trans-Plus-to-One-Plus21 2) . (ignore21 1))
   output
    ((Trans-Plus-to-One-Plus21 3) . (one-plus21 2)))
  :att-transfer-specs
    ((one-plus21 control-env
                 (control-env (nt-n> Trans-Plus-to-One-Plus21)))
     (one-plus21 feeds-back
                 (feeds-back (one-plus21 2) (one-plus21 1))))
  :transformation TRANSFORMATION-ONLY
  :grammars (the-grammar))
```

Bibliography

[1] Adam, Anne, and Jean-Pierre Laurent, "LAURA, A System to Debug Student Programs", *Artificial Intelligence*, 15 (1980), pp. 75-122.

[2] Brooks, Ruven, "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, 18 (1983), pp. 543-554.

[3] Brotsky, Daniel C., "Program Understanding Through Cliche Recognition", (M.S. Proposal), MIT/AI/WP-224, December, 1981.

[4] Brotsky, Daniel C., "An Algorithm for Parsing Flow Graphs", (M.S. Thesis), MIT/AI/TR-704, March, 1984.

[5] Bunke, Horst, "Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-4, No. 6, Nov., 1982.

[6] Bunke, Horst, "Graph Grammars as a Generative Tool in Image Understanding" in Ehrig, H., Nagl, M., Rozenberg, (Eds.): Proceedings 2nd Int. Workshop on Graph Grammars and their Application to Computer Science, Haus Ohrbeck, Germany, Oct. 4-8, 1982.

[7] Bunke, Horst, "Programmed Graph Grammars" in Lecture Notes in Computer Science, Vol. 56, 1977, pp. 155-166.

[8] Cyphers, D. Scott, "Automated Program Description", (B.S. Thesis), Working Paper 237, August, 1982.

[9] Cyphers, D. Scott, "Programming Cliches and Cliche Extraction", Working Paper 223, February, 1982.

[10] Earley, J., "An Efficient Context-Free Parsing Algorithm," (Ph.D. Thesis), Computer Science Department, Carnegie-Mellon University, 1968.

[11] Farrow, R., K. Kennedy, and L. Zucconi, "Graph Grammars and Global Program Data Flow Analysis", proceedings 17th Annual IEEE Symp. on Foundations of Computer Science, Houston, Texas, 1976.

[12] Faust, Gregory G., "Semiautomatic Translation of COBOL into HIBOL", (M.S. Thesis), MIT/LCS/TR-256, February, 1981.

[13] Fickas, Stephen and Ruven Brooks, "Recognition in a Program Understanding System", IJCAI-79, Tokyo, 1979, pp. 266-268.

[14] Genesereth, Michael R., "The Role of Plans in Intelligent Teaching Systems", Stanford, in D. Sleeman and J.S. Brown (Eds.): *Intelligent Tutoring Systems*, New York: Academic Press, 1982.

[15] Hall, Robert J., "On Using Analogy to Learn Design Grammar Rules", (M.S. Thesis), MIT/AI 1985.

[16] Johnson, W. Lewis, and Elliot Soloway, "PROUST: Knowledge-Based Program Understanding", IEEE Seventh Conference on Software Engineering, Orlando, Florida, 1984, pp. 369-386.

[17] Johnson, W. Lewis, and Elliot Soloway, "Intention-Based Diagnosis of Programming Errors", proceedings AAAI-84, Austin, Texas, August, 1984, pp. 162-168.

[18] Kennedy, Ken and Linda Zucconi, "Applications of a Graph Grammar for Program Control Flow Analysis", proceedings 4th Principles of Programming Languages, Santa Monica, 1977, pp. 72-85.

[19] Laubsch, Joachim, and Marc Eisenstadt, "Domain Specific Debugging Aids for Novice Programmers", The Open University, IJCAI, Canada, 1981, pp. 964-969.

[20] Lukey, F. J., "Understanding and Debugging Programs", *International Journal of Man-Machine Studies*, 12 (1980), pp. 189-202.

[21] Lutz, Rudi, "Diagram Parsing — A New Technique for Artificial Intelligence", University of Sussex, CSRP.054, 1986.

[22] Lutz, Rudi, "Program Debugging by Near-miss Recognition and Symbolic Evaluation", University of Sussex, CSRP.044, 1984.

[23] Miller, Mark L., "A Structured Planning and Debugging Environment for Elementary Programming", in D. Sleeman and J.S. Brown (Eds.): *Intelligent Tutoring Systems*, New York: Academic Press, 1982.

[24] Muchnick, Steven, and Neil Jones, "Program Flow Analysis: Theory and Applications", Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

[25] Murray, William R., "Automatic Program Debugging for Intelligent Tutoring Systems", The University of Texas at Austin, AI-TR86-27, June, 1986.

[26] Murray, William R., "Heuristic and Formal Methods in Automatic Program Debugging", proceedings IJCAI, LA, Calif., August, 1985.

[27] Rich, Charles, "A Formal Representation for Plans in the Programmer's Apprentice", proceedings IJCAI-81, August, 1981.

[28] Rich, Charles, "Inspection Methods in Programming", (Ph.D. Thesis), MIT/AI/TR-604, June, 1981.

[29] Rich, Charles, "Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too", Proceedings AAAI-82, August, 1982.

[30] Rich, Charles, "The Layered Architecture of a System for Reasoning about Programs", proceedings IJCAI-85, August, 1985.

[31] Rich, Charles, and Howard E. Shrobe, "Initial Report on a LISP Programmer's Apprentice", (M.S. Thesis), MIT/AI/TR-354, December, 1976.

[32] Rich, Charles, Richard C. Waters, and Howard E. Shrobe, "Overview of the Programmer's Apprentice", Proceedings of IJCAI-79, August, 1979.

[33] Rich, Charles and Richard C. Waters, "Abstraction, Inspection, and Debugging in Programming", MIT AI Memo No. 634, June, 1981.

[34] Ruth, Gregory R., "Analysis of Algorithm Implementations", (Ph.D. Thesis), MAC-TR-130, May, 1974.

[35] Shapiro, Daniel G., "Sniffer: a System that Understands Bugs", (M.S. Thesis), MIT AI Memo No. 638, June, 1981.

[36] Shrobe, Howard E., Richard C. Waters, and Gerald J. Sussman, "A Hypothetical Monologue Illustrating the Knowledge Underlying Program Analysis", A.I. Memo 507, January, 1979.

[37] Shrobe, Howard E., "Dependency Directed Reasoning in the Analysis of Programs Which Modify Complex Data Structures", Proceedings of IJCAI, August, 1979.

[38] Soloway, Elliot, Eric Rubin, Beverly Woolf, Jeffrey Bonar, and W. Lewis Johnson, "MENO-II: An AI-Based Programming Tutor", Yale RR-258, December, 1982.

[39] Steele, Guy L., Jr., *Common Lisp*, Digital Press, 1984.

[40] Vessey, Iris, "Expertise in Debugging Computer Programs: A Process Analysis", *International Journal of Man-Machine Studies*, 23 (1985), pp. 459-494.

[41] Waters, Richard C., "Automatic Analysis of the Logical Structure of Programs", (Ph.D. Thesis), MIT/AI/TR-492, December, 1978.

[42] Waters, Richard C., "A Method for Analyzing Loop Programs", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May, 1979.

[43] Waters, Richard C., "Program Translation via Analysis and Reimplementation", submitted to *IEEE Transactions on Software Engineering*, October, 1985.

[44] Waters, Richard C., "KBEmacs: A Step Toward the Programmer's Apprentice", MIT/AI/TR-753, May, 1985.

[45] Waters, Richard C., "The Programmer's Apprentice: A Session with KBEmacs", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, October, 1985.

CS-TR Scanning Project
Document Control Form

Date : 10/26/95

Report # AI-TR-904

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
□ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) □ Technical Memo (TM)
□ Other:__________

Document Information

Number of pages: 201(208-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
□ Double-sided

Intended to be printed as :
□ Single-sided or
☒ Double-sided

Print type:
□ Typewriter □ Offset Press ☒ Laser Print
□ InkJet Printer □ Unknown □ Other:__________

Check each if included with document:

☒ DOD Form (2) □ Funding Agent Form ☒ Cover Page
□ Spine □ Printers Notes □ Photo negatives
□ Other: __________

Page Data:

Blank Pages(by page number):__________

Photographs/Tonal Material (by page number):__________

Other (note description/page number):

Description : Page Number:
UN#
IMAGE MAP! (1-201) UN#'ED TITLE PAGE, BLANK PAGE, 2, UN#'ED BLK, 3,
UN#BLK, 4-198
(202-208) SCANCONTROL, COVER, DOD(2), TRGT'S(3)

Scanning Agent Signoff:
Date Received: 10/26/95 Date Scanned: 11/9/95 Date Returned: 11/16/95

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-904	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER AD-A186421
4. TITLE (and Subtitle) Automated Program Recognition		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Linda M. Wills		8. CONTRACT OR GRANT NUMBER(s) ONR N00014-85-K-0214 NSF DCR-8117633
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE February 1987
		13. NUMBER OF PAGES 202
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

analysis by inspection
computer-aided instruction
graph grammars
parsing
programming tutor
Programmer's Apprentice
Plan Calculus
program recognition
program understanding

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The key to understanding a program is recognizing familiar algorithmic fragments and data structures in it. Automating this recognition process will make it easier to perform many tasks which require program understanding, e.g., maintenance, modification, and debugging. This report describes a recognition system, called the Recognizer, which automatically identifies occurrences of stereotyped computational fragments

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (Abstract continued)
and data structures in programs. The Recognizer is able to identify these familiar fragments and structures, even though they may be expressed in a wide range of syntactic forms. It does so systematically and efficiently by using a parsing technique. Two important advances have made this possible. The first is a language-independent graphical representation for programs and programming structures which canonicalizes many syntactic features of programs. The second is an efficient graph parsing algorithm.