IMPLEMENTATION OF DATA ABSTRACTIONS VIA AN EVENT BASED PROGRAM TRANSFORMATION TECHNIQUE

ROBERT BLAINE BOYD BRAKE

Implementation of Data Abstractions Via An Event Based Program Transformation Technique

By

© Robert Blaine Boyd-Brake, B.Sc.

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science

Department of Computer Science
Memorial University of Newfoundland

November 1985

St. John's Newfoundland

ISBN 0-315-31002-2

Abstract

Conventional data abstraction techniques enhance several aspects of a software system, its developement, maintainability, reusability and comprehensibility. However, using conventional techniques one must manually implement a data abstraction and it's operators, which may result in an inefficient implementation.

This paper introduces a program tranformation technique which mechanically implements a fixed data abstraction with a limited set of efficient implementations.

Within the limited scope, the event based technique has all the advantages of the conventional methods, as well as, automatic and efficient implementation by using an inline coding technique which is based on the properties (attributes) of the data representation being used to implement the data abstraction.

Acknowledgements

I would like to thank Dr. A. G. Middleton for being my advisor on this thesis and for providing the insight to narrow the problem to one of manageable porportions...

I also would like to thank Memorial University of Newfoundland for providing the funding for this thesis through a Graduate Fellowship and a Teaching Assistantship.

Table of Contents

List of Tables

List of Figures

1. Introduction

The concept of data abstraction is widely accepted as a useful paradigm in software design. Its use allows a programmer to clarify his view of a software system by the supression of *irrelevant details* [13,14,27]. However, the concept of data abstraction does nothing to *support directly* the generation of the program code required to provide the *implementation details;* this must be performed *manually* by the programmer.

Methodologies have been developed that have proven useful in the design and implementation of data abstractions:

(i) **Procedural Implementatation of the data abstraction and its set of operations.** Conventional programming languages, such as Pascal, C and Fortran, have been used as the implementation languages of major software systems. In these languages, the subprogram (function/procedure) mechanism is commonly used when implementing operations that are to be used in many places, such as data abstraction operations.

(ii) **Encapsulation of design decisions.** A design decision is implemented by providing a collection of subprograms (modules). These subprograms can be invoked from other subprograms, but their *implementation details* are hidden from the rest of the system [27]. This is to prevent incorrect use of these modules.

(iii) **Hierarchical structuring of modules.** The modules or components that make up the software system are implemented using a hierarchical structure. The software system is created by implementing low level modules which are integrated into the next higher level of the system and so on. This helps clarify the system and allows the programmer to move easily from one level of abstraction to another.

The above techniques, at the implementation level, enhance the software's development, comprehensibility, reliability, maintainability, and reusability. However, in using the above techniques there are two problem areas:

(i) **Implementation of the data abstraction and its set of operations.** When given a general solution in terms of a data abstraction, the programmer must *manually* provide the implementation details. Depending on the structure used, this can be a trivial or non-trivial process.

(ii) **Efficiency of the resulting implementation.** Conventional techniques use a dynamic or subprogram approach to implement a data abstraction. Using the ideas of encapsulation and hierarchical structuring, further dynamic or subprogram levels are introduced. The implementation of a data abstraction and its associated set of operations can be efficiently implemented by most programmers. Inefficiency with respect to space and time arise, when these subprograms are *used* throughout the software system. A subprogram to perform an action will have subprogram calls to manipulate the data abstraction ; these subprograms may have further subprogram calls and so on. To perform any operation requires a large communication overhead [4,7].

This work is an investigation into the automation of implementing data abstractions in a conventional programming language ; the abstraction under consideration is the *string* or *sequence* [6,29,30]. The implementation derived should have the advantages of conventional data abstraction: development, comprehensibility, reliability, maintainability, and reusability. Furthermore, there should be *mechanical translation* and *efficient implementation* of the data abstraction.

Much of the mechanics of the system implemented is based on the use of an *event based program transformation technique* [26]. The diagram shown in Figure 1 represents a simplified description of the implementation of the system.

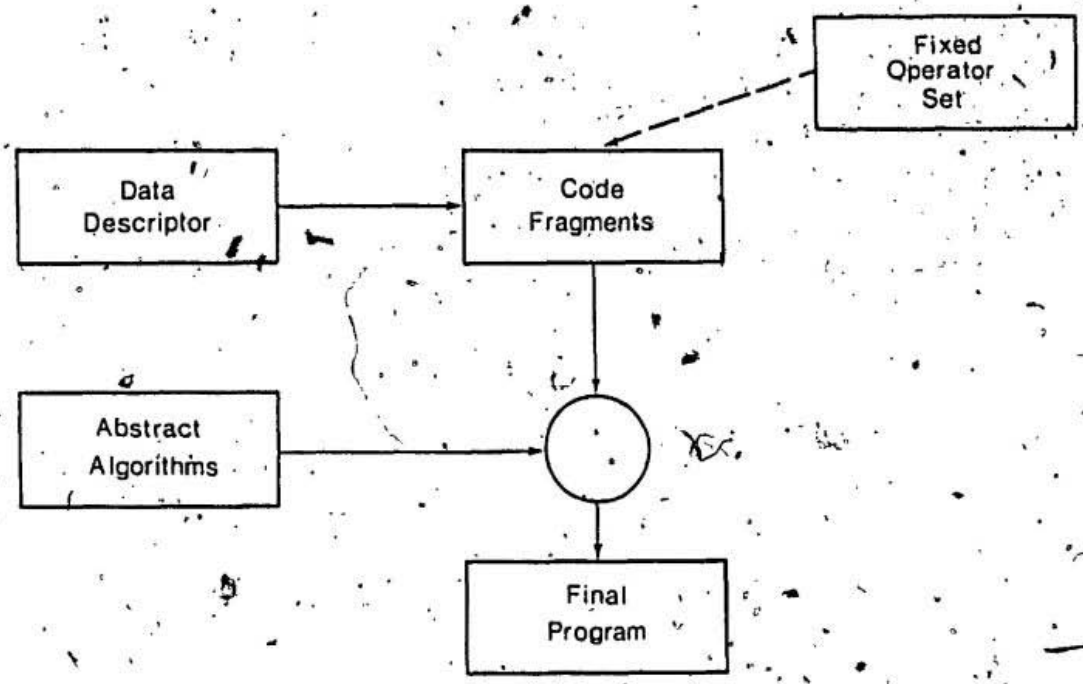


Figure 1 General structure of the system

The system uses a notation called an abstract algorithm to represent the action to be performed. The abstract algorithm contains events, which are embedded in the algorithm to act as markers to show where data abstraction operations are to be performed.

Once given a data descriptor of the structure to be used (array,record,list,ring,...), the system generates the code fragments. These code fragments represent the implementation code for the data abstraction operations (operator set) using the specified data structure.

The abstract algorithms are transformed into the final program by using the event based program transformation technique. The transformations take the form of text-for-text substitutions for the specific event and its implementation (code fragment), rather than the more commonly used tree-for-tree substitutions used in other transformation systems [18,23].

2. Survey

In the automatic programming field there are four main research efforts: high level languages, process model, transformation/program synthesis, and knowledge based programming.

2.1. High level languages

The term used to identify this section covers a broad range of languages. However, most of the research being performed can be classified under two headings: nonprocedural/functional programming and support of advanced programming features.

2.1.1. Nonprocedural/Functional programming

Languages developed using this approach are concerned with allowing the programmer to specify a goal to be achieved, rather than having him give a specific method of solution.

Many nonprocedural/functional languages have all or most of the following features:

(i) **Associative Referencing:** The ability to access data based on some intrinsic property of the data. This is important because the programmer does not have to specify access paths explicitly or program an algorithm to perform a search for a specific data structure [21].

(ii) **Aggregate Operators:** The ability to perform convoluted operations in a single step.

(iii) **Elimination of Arbitrary Sequencing:** The elimination of all sequencing *not required* by the data dependencies. In nonprocedural/functional programming, data dependencies are shown explicitly by the operand-operand structure of the

program. There is no assignment or goto, the outcome desired is specified as a function of inputs, rather than indicated by a step by step sequence of program steps.

(iv) **Nondeterministic Programming and Parallelism:** The ability to perform the following types of operations is required:

(a) A choice function which conceptually executes all paths in parallel, with each path having a choice as an argument.

(b) A success function as the termination of computation.

(c) A failure function as the termination of computation.

When performing a choice operation, only those paths that return a success are considered to be computations of the algorithms. Languages with the choice function may imply the use of the following: automatic backtracking, pattern directed data base searches and pattern directed invocation of functions.

Some languages which fall within the nonprocedural/functional classification are FP [2], SETL [21] and PROLOG [8].

2.1.2. Support of advanced programming features

This approach is concerned with moving from conventional type languages (procedural), such as Pascal and Fortran, to higher level languages (procedural) which *readily* support abstractions and other advanced features.

Alphard [35] and Clu [22] are two of many such languages. In these two languages a data abstraction is represented as a set of objects and a set of operations for manipulating these objects. The abstraction mechanism and the formal verification of programs is better supported by using Alphard's form and Clu's cluster to constrain the operations

that may be performed on the data abstraction and the encapsulation of the implementation details [22,35].

The introduction of new programming features, such as, strings [6,29,30] and concurrent programming [5] are also considered to fall within this area.

2.2. Transformation/Program synthesis

This approach is concerned with the derivation of efficient implementations of programs by a transformation process. The transformations commonly are production rule based. If the left hand of a production rule is matched, the right hand substitution is made for that occurrence. These production rules are normally concerned with optimizing the control abstractions (loops, selection, branching, ...) used [18,23] but there are others that are concerned with optimizing the data abstraction implementation [4,7].

The transformational process is used to eliminate the overhead commonly associated with modular design or structured programming. Efficiency is the major concern of this approach.

2.3. Process model

The basis of this approach is the relationship between data and the program structure. A *decomposition* of a program into *distinct* processes can be obtained by making the program and data structures used represent a *model* of the real world situation. The decomposition *directly reflects* real world behavior and thereby is easy to maintain and logically correct[15].

The Jackson design methodology [15] is appropriate only to the class of problems which are strongly and inherently sequential, such as data processing. The Jackson design method has been automated and is currently in use by many companies to perform their

data procesing needs [14].

2.4. Knowledge based programming

This approach is concerned with representing knowledge so that it can be used throughout an automated process to develop solutions to problems within the symbolic programming domain.

PSI [3] is such a system and was implemented by using two phases: an acquisition phase and a synthesis phase. The acquisition phase through a natural language dialogue and traces, develops a program skeleton of the solution to the problem. The synthesis phase takes the program skeleton as input. Using a rule based representation of knowledge and through the interaction of a coding expert (PECOS) [3] and an efficiency expert (LIBRA) [3,15], synthesizes Lisp programs which implement the program skeleton.

The PSI system has been used to solve the following problems: membership test, concept formulation, sorting, finding primes, reachability and simple classification [3].

3. Event Based Program Transformations

The event based program transformation technique presented in this paper falls under the Transformation/Program Synthesis heading of the previous section.

The technique uses an *abstract algorithm* notation to specify what action must be performed. The abstract algorithm is written in pseudo Pascal and contains markers (events) which are used to specify the type of operation that must be performed.

Consider a print algorithm which prints the contents of a sequence (string). The solution involves the iteration over the string and the printing of the contents of the current state of iteration until the end of string is determined.

The print algorithm is the following:

```
<INIT>
WHILE NOT (<EOF>) DO BEGIN
  WRITE(<NOW>)
  <MOVE>
END
WRITELN
```

with the following notation:

<INIT>	~	initialize for iteration over sequence
<EOF>	~	test for end of sequence
<NOW>	~	produce contents of current location in sequence
<MOVE>	~	move to next location in sequence

The print algorithm consists of a sequence of steps which will result in the contents of the string being printed. The events in the abstract algorithm are normally parameterized to name the objects to be manipulated.

The print algorithm becomes:

```
<INIT : SRC>
WHILE NOT (<EOF : STSRC>) DO BEGIN
  WRITE(<NOW : STSRC>)
  <MOVE : STSRC>
END
WRITELN
```

with the following notation:

(i) SRC is the source sequence to be printed.

(ii) The prefix ST denotes a state variable which is needed to access the source sequence and is used to indicate the current status of iteration.

The print algorithm contains the operations (events) that must be performed to solve the print problem. Depending on the actual data representation used, the program code (implementation) representing the algorithm can be quite different.

For example, if an array representation with the '0' symbol as end of sequence marker is used, the print algorithm becomes:

```
STSRC := 1;
WHILE NOT (SRC[STSRC] = '0') DO BEGIN
 WRITE(SRC[STSRC]);
 STSRC := STSRC + 1
END;
WRITELN
```

with the following declaration being generated:

```
VAR
   SRC   : ARRAY [1..10] OF CHAR;
   STSRC : INTEGER;
```

If a singly linked list with a pointer to the first element (SRC) is used as the data representation, then the following declaration would be generated:

```
TYPE
   LINK = ^BODY;
   BODY = RECORD
          DATA : CHAR;
          FWDPTR : LINK
         END;

VAR
   SRC : LINK;
   STSRC : LINK;
```

and the print algorithm would be transformed to the following:

```
STSRC := SRC;
WHILE NOT (STSRC = NIL) DO BEGIN
  WRITE(STSRC^.DATA);
  STSRC := STSRC^.FWDPTR
END;
WRITELN
```

The code fragments representing the events in the print algorithm are shown in Table 1. The events in the abstract algorithm act as markers in the abstract algorithm at which abstract operations are performed, such as, initialize for iteration over sequence, test for end of sequence, ... As shown in Table 1, the implementation code for the abstract operations invoked by the same event are significantly different. The implementation code is dependent upon the data representation in use.

Table 1 Code fragments representing the events in the print algorithm

Name	Array	Linked List
<INIT>	STSRC := 1	STSRC := SRC
<EOF>	SRC[STSRC] = '@'	STSRC^.FWDPTR = NIL
<NOW>	SRC[STSRC]	STSRC^.DATA
<MOVE>	STSRC := STSRC + 1	STSRC := STSRC^.FWDPTR

The above examples introduce the ideas of abstract algorithms (data representation independent) and events (operations). They show it is possible to specify a solution to a problem without having to consider specific data representations. When given a specific data representation, it is possible to transform the abstract algorithm so that it

implements the required process using the specified data representation.

The remainder of this paper is dedicated to providing an explanation of the the event based technique and answering some basic questions:

(i) How does the technique automatically generate programs which implement the sequence (string) abstraction?

(ii) How does the technique incorporate knowledge concerning the abstraction in use?

(iii) Are efficient implementations derived?

4. Automatic Code Generation

This section introduces the terminology associated with the event based program transformation technique and shows how the Pascal implementation is generated.

4.1. Abstract algorithms

An *abstract algorithm* is a notation used to describe a process to be performed. It is written in pseudo Pascal and contains *events* [26] which are used to indicate the points in the algorithm at which abstract operations must be performed. The events (operations) which a typical general algorithm could contain are: initialize for iteration over sequence, test for end of sequence, produce contents of current location in sequence,... . A list of the basic events available can be seen in Table 2.

Table 2 Basic events in the abstract algorithms

Name	Description
<INIT>	initialize for iteration over sequence
<EOF>	test for end of sequence
<NOW>	produce contents of current location in sequence
<MOVE>	move to next location in sequence
<CLEANUP>	perform cleanup operation at the end of iteration

For example, an algorithm which will *convert* from a *source* representation (SRC) to a *destination* representation (DST) is the following:

```
<INIT : SRC>
<INIT : DST>
WHILE NOT (<EOF : STSRC>) DO BEGIN
```

```
    <NOW : STDST> := <NOW : STSRC>
    <MOVE : STSRC>
    <MOVE : STDST>
  END
  <CLEANUP : STSRC>
  <CLEANUP : STDST>
```

With the following notation:

(i) SRC is the *source sequence* of the convert and DST is the *destination sequence* which receives a copy of the data contained in the SRC.

(ii) The prefix ST denotes a *state variable* which is needed to access the source or destination sequence and is used to indicate the current status of iteration.

It was found more convenient to represent the abstract algorithms as a procedure or function, as below:

```
PROCEDURE CONVERT ( <PARAM> )
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  <INIT : DST>
  WHILE NOT ( <EOF : STSRC> ) DO BEGIN
    <NOW : STDST> := <NOW : STSRC>
    <MOVE : STSRC>
    <MOVE : STDST>
  END
```

```
<CLEANUP : STSRC>
<CLEANUP : STDST>
END
```

In the convert algorithm there are two new events introduced, <PARAM> and <LOC_VAR> which stand for parameter list and local variable declarations respectively. A complete list of all events can be found in Table 3.

Table 3 Events available for use in the abstract algorithms

Name	Description
<INIT>	initialize for iteration over sequence
<EOF>	test for end of sequence
<NOW>	produce contents of current location in sequence
<MOVE>	move to next location in sequence
<CLEANUP>	perform cleanup operation at the end of iteration
<ACCESS_SIZE>	access size of sequence
<ACCESS_LAST>	access last location in sequence
<PARAM>	generate parameter declaration for basic algorithm
<LOC_VAR>	generate local variable declaration
<NULL_ADD>	generate empty sequence representation
<FN_TYPE>	generate function type required by data representation

The events shown in Table 3 represent a small but useful subset of the abstract operations that can be performed on the sequence (string) abstraction. Using the events

shown in Table 3 the abstract algorithms in Table 4 were implemented and represent a *minimal* set of routines for manipulating the sequence (string) abstraction. The actual representations of the abstract algorithms can be seen in Appendix A and Appendix B.

Table 4 Abstract algorithms for representing the string abstraction

Name	Description
APPEND	append one string to another
CONVERT	convert from one representation to another
COPY	copy a string (special case of convert)
DELETE	delete a substring, given its position in a parent string and its length
EMPTY	produce the empty string
EXTRACT	extract a substring, given its position in a parent string and its length
FIND	search for a substring within a parent string
INDEXOF	index the string (as a vector), ie. give address of fifth character in the string
INSERT	insert one string in another
PRINT	print a string
PUTSTRING	add a single element to the end of a string
RELATION	compare strings lexicographically (one of EQ,NE,LT,LE,GT,GE supplied as a parameter)
SIZE	return size of string
SHIFT	shift elements left or right in an array representation (internal algorithm, not available to the user)

4.2. Events

The events shown in the previous section were used in the abstract algorithms to show where a certain type of *operation* should take place.

Consider the CONVERT algorithm from the previous section.

```
PROCEDURE CONVERT ( <PARAM> )
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  <INIT : DST>
  WHILE NOT ( <EOF : STSRC> ) DO BEGIN
    <NOW : STDST> := <NOW : STSRC>
    <MOVE : STSRC>
    <MOVE : STDST>
  END
  <CLEANUP : STSRC>
  <CLEANUP : STDST>
END
```

The abstract algorithm is specified in terms of the string (sequence) abstraction. The actual data representation used for the string or sequence affects the code fragments generated to represent the events (operations). Consider the conversion from an array representation (SRC) to a singly linked list representation (DST). Figure 2 shows a simple example.

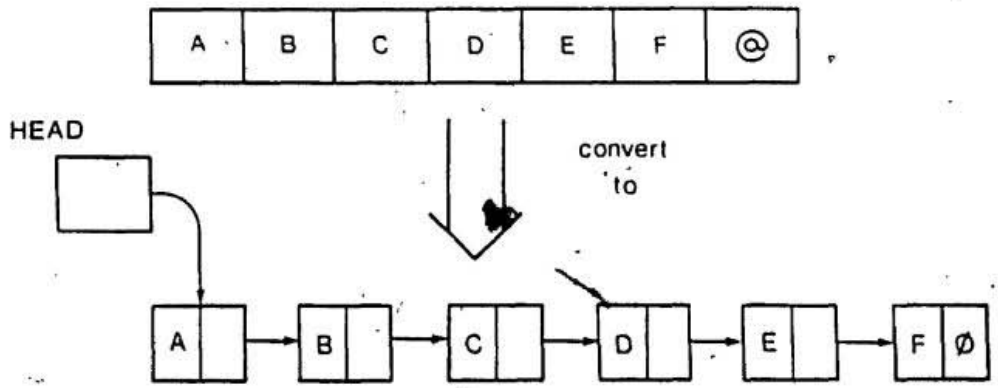


Figure 2 Conversion of an array representation to a singly linked list representation

where *HEAD* is a pointer to the *first* cell in the linked list and the '@' is used to mark the end of sequence for the array representation.

The following type declarations would be generated:

```
TYPE

  NAME = ARRAY[1..7] OF CHAR;

  LINK = ^BODY;
  BODY = RECORD
         DATA : CHAR;   (* character field *)
         FWDPTR : LINK  (* forward pointer *)
       END;
```

The following code fragments representing the events would be generated:

<PARAM>	⟹	SRC : NAME; VAR DST : LINK
<LOC_VAR>	⟹	VAR STSRC : INTEGER; PREVDST,STDST : LINK;
<INIT : SRC>	⟹	STSRC := 1;
<INIT : DST>	⟹	NEW(STDST); DST := STDST;

```
                                    PREVDST := NIL;

<EOF : STSRC>          ⟹           SRC[STSRC] = '@'

<NOW : STSRC>          ⟹           SRC[STSRC]

<NOW : STDST>          ⟹           STDST^.DATA

<MOVE : STSRC>         ⟹           STSRC := STSRC + 1

<MOVE : STDST>         ⟹           PREVDST := STDST;
                                    NEW(STDST);
                                    PREVDST^.FWDPTR := STDST

<CLEANUP : STSRC>      ⟹           (* EMPTY SEQUENCE *)

<CLEANUP : STDST>      ⟹           IF PREVDST = NIL THEN
                                       DST := NIL
                                    ELSE
                                       PREVDST^.FWDPTR := NIL;
                                    DISPOSE(STDST)
```

(An explanation of how the system generates the code fragments which represent the events will be given in Section 6.0.)

The code fragments generated by most of the events should be obvious from the data representation in use and the type of operation being performed. The code for the event <CLEANUP : STSRC> is not obvious; the code fragment generated is the empty sequence. In the CONVERT algorithm the array representation (SRC) is passed as a value parameter and the algorithm is performing a *non-destructive* iteration over the array representation, therefore there is no need for a cleanup operation and the empty sequence is generated.

4.3. Code generation

The previous sections 4.1 and 4.2 have introduced the ideas of an abstract algorithm and events.

How does the technique generate the program implementation given the abstract algorithm and the code fragments representing the events?

The transformation process is production rule based and takes the form of text substitution for the specific instance of an event.

Given the abstract algorithm:

```
PROCEDURE CONVERT ( <PARAM> )
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  <INIT : DST>
  WHILE NOT ( <EOF : STSRC> ) DO BEGIN
    <NOW : STDST> := <NOW : STSRC>
    <MOVE : STSRC>
    <MOVE : STDST>
  END
  <CLEANUP : STSRC>
  <CLEANUP : STDST>
END
```

Using the code fragments shown in section 4.2, the abstract algorithm is transformed by the inline coding of the code fragments representing the events.

The final Pascal implementation after the inline coding action is the following:

```
PROCEDURE CONVERT (SRC : NAME; VAR DST : LINK);
VAR
   STSRC : INTEGER;
```

```
    PREVDST,STDST : LINK;
BEGIN
    STSRC := 1;

    NEW(STDST);
    DST := STDST;
    PREVDST := NIL;

    WHILE NOT (SRC[STSRC] = '0') DO BEGIN

      STDST^.DATA := SRC[STSRC];

      STSRC := STSRC + 1;

      PREVDST := STDST;
      NEW(STDST);
      PREVDST^.FWDPTR := STDST

    END;

    IF PREVDST = NIL THEN
      DST := NIL
    ELSE
      PREVDST^.FWDPTR := NIL;
    DISPOSE(STDST)

END; (* CONVERT *)
```

The above is the Pascal implementation of the conversion algorithm, using an array (SRC) and a singly linked list (DST).

5. Choice of Representations

In Pascal there are two basic types of data structures available to represent a string: a static and a dynamic data structure. The static structures involve the allocation of storage for the string at compile time, examples are arrays and user-defined static record structures. The dynamic structures involve the allocation of storage at run time, examples are any user-defined dynamic structures: lists, rings, trees, ...

The string representations that were allowed in the system are the following:

(i) Optionally, a string can have a *header* which could contain a nonempty subset of the following:

  (a) The position of the first element of the string. This is a compulsory component unless the the header and the data body coexist in the same storage aggregate.

  (b) The position of the last element in the string.

  (c) The size of the string.

(ii) All strings have a data *body* which contains the actual elements of the string. The following representations are allowed for the data body:

  (a) Array

  (b) Singly Linked List

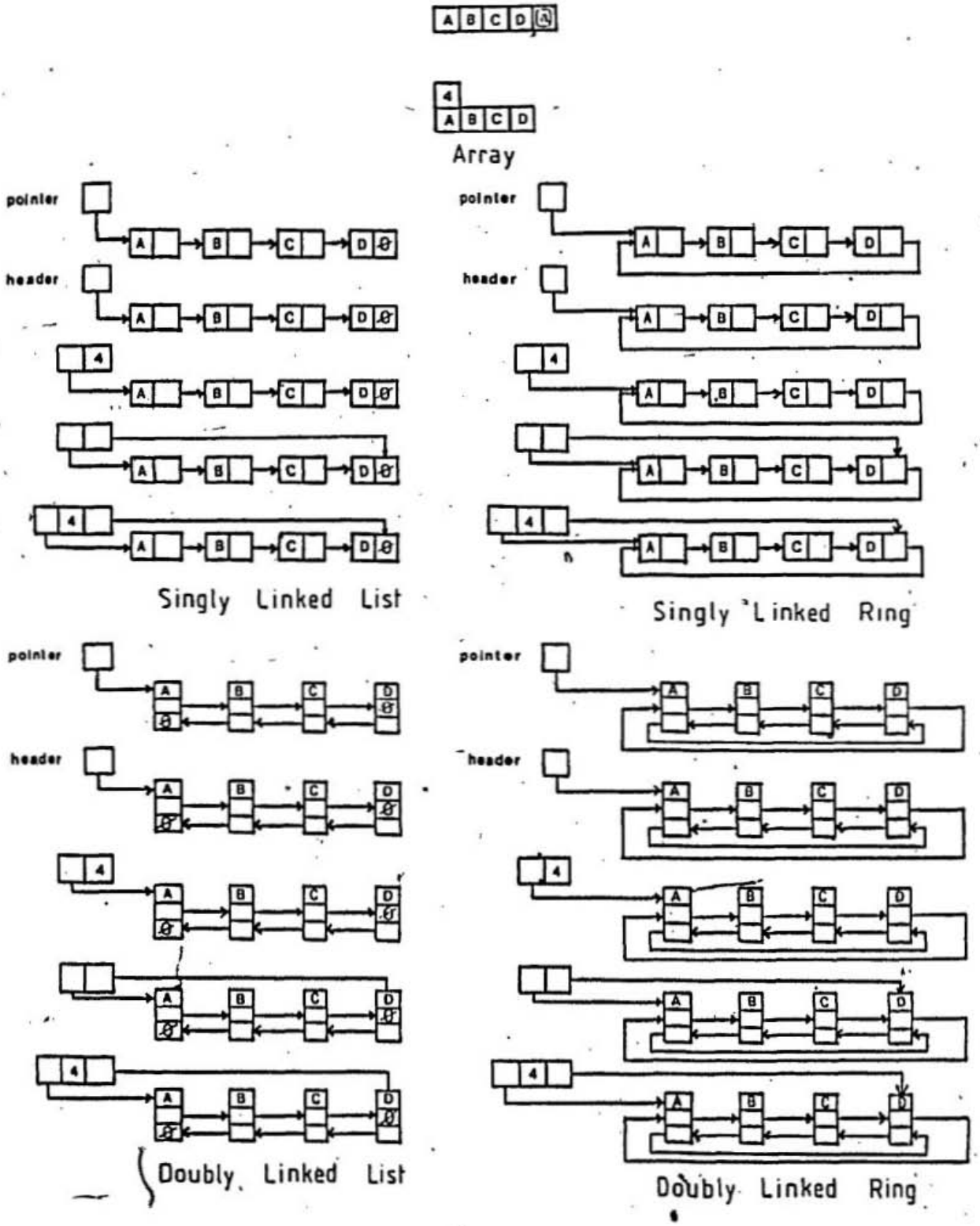
  (c) Doubly Linked List

  (d) Singly Linked Ring

  (e) Doubly Linked Ring

Figure 3 contains a graphical view of the available string representations.

Figure 3 Avaliable string representations



The above is far from a definitive collection of possible string representations, but is adequate to make the central points contained in this paper. The use of conventional data abstaction techniques does nothing to support directly the generation of the program code required to provide the implementation details; this must be performed manually by the programmer. Depending on the data structures used, this can be a trivial or non-trivial task. Even if the data structure is trivial it is still a tedious process where the programmer has to concern himself with a myriad of miscellaneous information. It is far better for the programmer to develop a solution to the problem (algorithm) and let the implementation details be provided by an automated process.

6. Derivation of Program Code for an Event

The events in the abstract algorithms are used as markers to drive the transformation process. The transformations take the form of an inline coding of the code fragments representing the operations invoked by the events.

This section introduces the idea of decision trees and shows how they can be used to look after both compile-time and run-time operations required by the implementation of the abstract algorithm.

6.1. Decision trees / conditional compilation

Conditional compilation is a mechanism used in some programming languages to *switch* on or off various statements in a program. This is useful in that it allows the same program to be run on different computer systems, aids debugging and tracing of the program, ... The conditional compilation process can be represented as an explicit sequence of steps or a graphical representation (decision trees).

The *decision tree* graphical representation allows a user to visualise easily the conditional compilation process. The decision tree is a tree representation consisting of ⊕ concatenation nodes, ? decision nodes and leaves which correspond to program code. Figure 4 shows a simple decision tree.

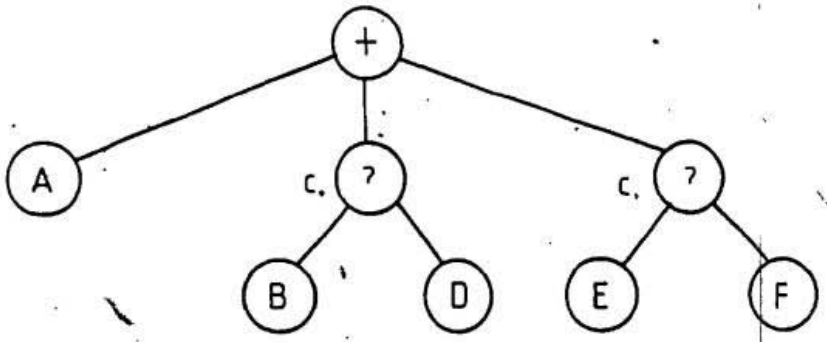


Figure 4 Decision tree for code generation

with the following notation:

⊕	concatenation of code fragments
?	choice between different code fragments
c_i	condition which must be tested at a decision node
A,B,D,E,F	program code which is used to build the code fragments which represent the events

The code fragments are generated by a post order traversal of the decision tree, where at a decision node ? , the left subtree is traversed if the condition c_i is true, otherwise the right subtree is traversed.

For example, in the decision tree shown in Figure 4, if the conditions c_0 and c_1 were both true ($c_0 c_1$), the decision tree in Figure 4 would reduce to the following:

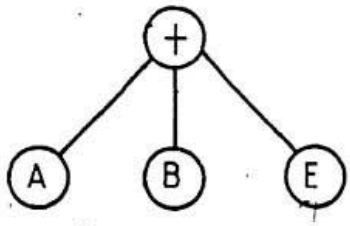


Figure 5 Reduced decision tree $c_0 c_1$

After the concatenation is performed, the code generated is ABE.

If in the decision tree shown in Figure 4 the conditions c_0 and not c_1 were true ($c_0 \neg c_1$), the decision tree shown in Figure 4 would reduce to the following:

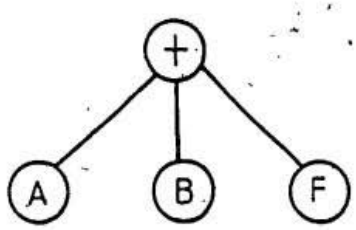


Figure 6 Reduced decision tree $c_0 \neg c_1$

After concatenation is performed, the code generated is ABF. Likewise, if the conditions not c_0 and c_1 were true ($\neg c_0 c_1$), after the concatenation process the code ADE would be generated. If the conditions not c_0 and not c_1 were true ($\neg c_0 \neg c_1$), after the concatenation process the code ADF would be generated.

The above examples show, by testing if a condition (c_i) is true or false and using a concatenation process, the decision tree generates code fragments which are appropriate for the given values of c_i.

6.2. Attributes

For this project, conditions in the decision tree depend on the *attributes (properties)* of the data representation in use and the particular event.

To determine the code representing an event involves the following decisions:

(i) What type of event is being derived? Each type of event, whether <INIT>, <MOVE>, <NOW>, <EOF>, ..., involves different decisions which depend upon the attributes of the particular event.

(ii) What type of data representation is being used to represent the data abstraction? Each data representatation has attributes which distinguish it:

  (a) Static or dynamic representation.

  (b) Type of iteration:

    (1) Nondestructive.

    (2) Type of destructive iteration:

      (i) Insertion.

      (ii) Deletion.

  (c) Any attributes which need special consideration:

    (1) The size of a string.

    (2) The position of the last element in the string.

    (3) The position of the first element in the string.

6.3. Operator set events and overhead events

An *operator set* event is an event whose associated operation is performed at run-time. Examples of operator set events are: initialize for iteration over sequence, test for

end of sequence, ... Table 5 contains a list of all events which are called operator set events.

Table 5 Operator set events

Name	Description
<INIT>	initialize for iteration over sequence
<EOF>	test for end of sequence
<NOW>	produce contents of current location in sequence
<MOVE>	move to next location in sequence
<CLEANUP>	perform cleanup operation at the end of iteration
<ACCESS_SIZE>	access size of sequence
<ACCESS_LAST>	access last location in sequence

Overhead events are events whose operation is performed at compile-time. Examples of overhead events are: parameter specifications, local variable declarations, function type declarations,... . Table 6 contains a list of the events which are called overhead events.

Table 6 Overhead events

Name	Description
<PARAM>	generate parameter declaration for abstract algorithm
<LOC_VAR>	generate local variable declaration
<NULL_ADD>	generate empty sequence representation
<FN_TYPE>	generate function type required by data representation

Operator set events are handled easily by conventional data abstraction techniques by using the subprogram (procedure/function) mechanism , however, there is a large dynamic overhead associated with such an implementation. Overhead events cannot be handled easily by conventional data abstraction techniques because compile-time operations are not supported by conventional languages (no compile-time functions).

In the methodology presented in this project, *overhead events* and *operator set events* are handled by the same (decision tree) mechanism. The methodology is used to generate *text* , which happens to be the implementation code of an abstract algorithm using a particular data representation. The text generated, is a program segment which handles both compile-time (overhead events) and run-time (operator set) aspects of code generation, this being a requirement for automatic program generation.

6.4. Derivation of the implementation code using a decision tree

The derivation of the implementation code representing an event depends upon the attributes of the data representation and the event. Different operators depend on different properties (attributes) of the data representation (not every attribute affects every operator).

The easiest way of showing the derivation of the code representing an event is to consider an example. Consider the CONVERT example of Section 4.0, the conversion from an array representation (SRC) to a singly linked list representation (DST) without a header cell.

The following type declaration is generated:

```
TYPE
   NAME = ARRAY[1..7] OF CHAR;

   LINK = ^BODY;
   BODY = RECORD
        DATA : CHAR;  (* data field *)
        FWDPTR : LINK  (* forward pointer *)
      END;
```

The system then considers the abstract algorithm for the CONVERT:

```
PROCEDURE CONVERT ( <PARAM> )

 <LOC_VAR>

BEGIN

 <INIT : SRC>

 <INIT : DST>

 WHILE NOT ( <EOF : STSRC> ) DO BEGIN

  <NOW : STDST> := <NOW : STSRC>

  <MOVE : STSRC>

  <MOVE : STDST>

 END

 <CLEANUP : STSRC>

 <CLEANUP : STDST>

END
```

The events are extracted from the abstract algorithm; Table 7 contains a list of the events in the convert algorithm that need their code fragments generated. Each event is coded using attributes deduced from the data descriptor and the abstract algorithm.

Table 7 Events used in the convert algorithm

Event	Description
<PARAM>	parameter declaration for abstract algorithm
<LOC_VAR>	local variable declaration for abstract algorithm
<INIT : SRC>	initialize source sequence for iteration
<INIT : DST>	initialize destination sequence for iteration
<EOF : STSRC>	test for end of source sequence
<NOW : STSRC>	produce contents of current state of source sequence iteration
<NOW : STDST>	produce contents of current state of destination sequence iteration
<MOVE : STSRC>	move to next location in source sequence
<MOVE : STDST>	move to next location in destination sequence
<CLEANUP : STSRC>	perform clean up operation required for source sequence
<CLEANUP : STDST>	perform clean up operation required for destination sequence

A decision tree structure is used to derive the code fragment representing an event. Figure 7 shows the decision tree involved in deriving the program code representing the <INIT> event in the convert algorithm.

The decisions involved are basically the following:

(i) The type of data structure: dynamic (run time allocation of storage) or static (compile time allocation of storage). This information is determined from the descriptor of the data structure [1].

1 The system when given the data representation to be used, represents this in a internal form called a data descriptor. See Appendix C.

(ii) The type of iteration: destructive (change made to the values stored or representation) or nondestructive. This information is determined from the type of routine: Destructive (APPEND, CONVERT, COPY, ...) or Nondestructive (EMPTY, FIND, INDEXOF, RELATION, ...).

(iii) Attributes which need special consideration: size or header cell. This information is determined from the data descriptor.

The decision tree shown in Figure 7 is used to generate the code fragment through a concatenation and selection process based on the presence or absence of required attributes (properties) [2]. It can be seen by inspection of Figure 7, that the generation of the code fragment for the <INIT> event can depend on the following attributes:

(i) Is a static or dynamic data representation being used?

(ii) Is destructive (change of values stored or representation) or nondestructive iteration being performed?

(iii) Is a header cell present?

(iv) Is a size present or must one be computed?

It can be seen by observation, that even a very simple event such as <INIT> (initialize for iteration over sequence), can involve quite a few of the attributes of the data representation and the abstract algorithm.

2 The actual implementation is a list structure with list concatenation

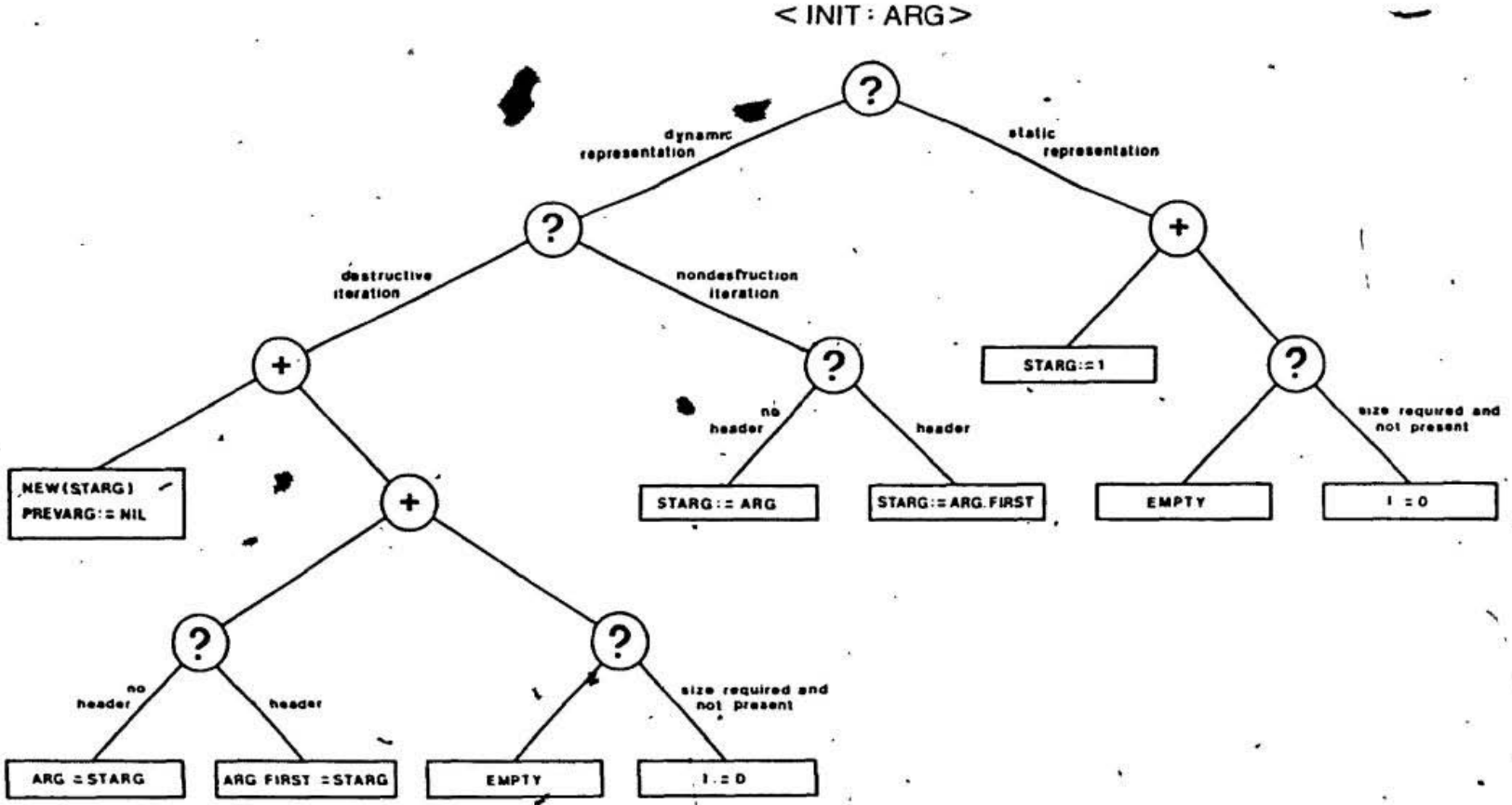


Figure 7 Decision tree involved in generating code fragments which represent the <INIT> event.

? - decision

+ - concatenation - post order traversal

The derivation of the <INIT : SRC> event follows a path as shown in Figure 7.

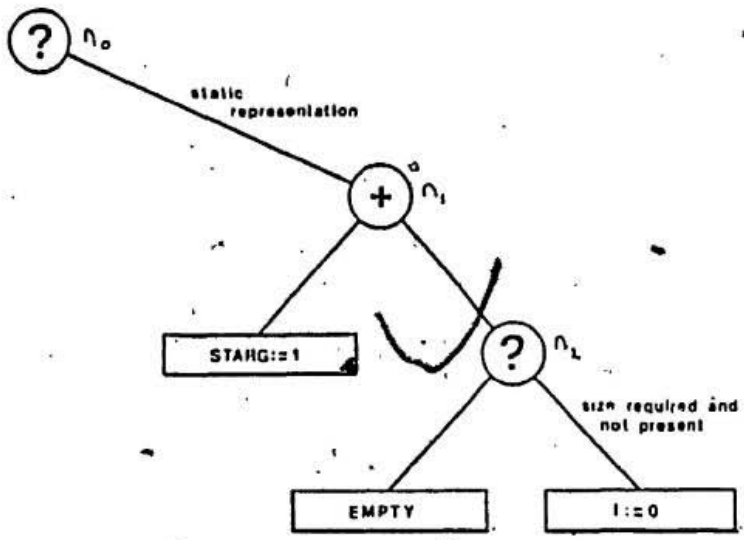


Figure 8 Derivation of the code fragment representing <INIT : SRC> event in the convert algorithm

where the n_i are used as labels to aid in the explanation of the code derivation process.

The derivation of the code fragment which represents the <INIT : SRC> event starts at the root of the decision tree shown in Figure 8, node n_0. The condition tested at the decision node n_0 is whether a static or dynamic representation is being used. From the data descriptor, it is determined that an array representation is being used, and as such, is a static representation. The right path from the node n_0 is traversed.

The traversal of the right path leads to the concatenation node n_1 (post order traversal). The left path of the node n_1 is traversed and a leaf is met. At the leaf

the code fragment

STARG := 1

is found. The decision tree is traversed upward until the concatenation node n_1 is met again. The right path at the concatenation node n_1 is then traversed.

The traversal of the right path leads to the decision node n_2. The condition tested at the decision node n_2 is whether a size is present or required. From the data descriptor it is determined that no size is required and the left path of the node n_2 is traversed. A leaf is met and it contains the *empty* code fragment. The empty code fragment is used when no statement needs to be generated.

The decision tree shown in Figure 8 reduces to:

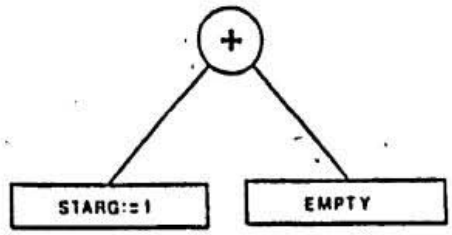


Figure 9. Reduced decision tree required by <INIT : SRC>

After the concatenation process and the substitution of the string "SRC" for the string "ARG", the following code fragment is generated:

<INIT : SRC> $\Longrightarrow$ STSRC := 1

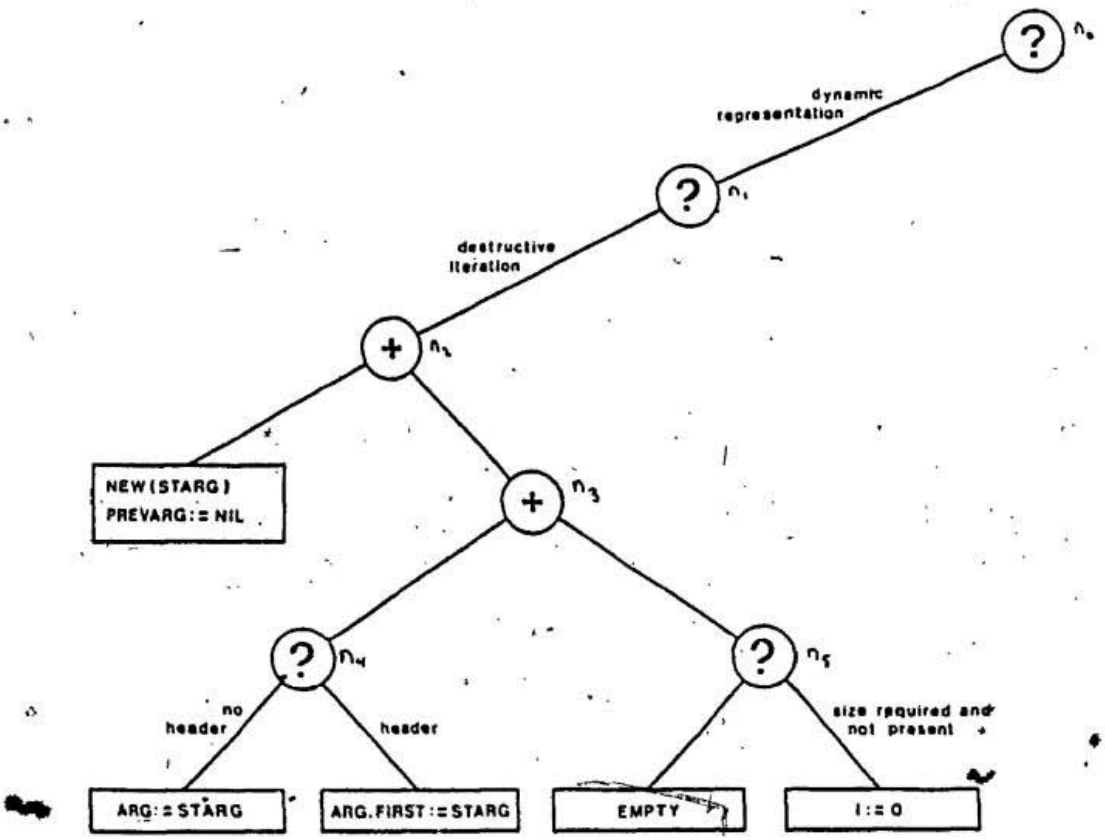


Figure 10 Derivation of the code fragment representing <INIT : DST> event in the convert algorithm

The derivation of the code fragment representing the event <INIT : DST> starts at the root of the decision tree shown in Figure 10, node n_0. The condition tested at the decision node node n_0 is whether a static or dynamic representation is being used. From the data descriptor it is determined that a singly linked list representation without a header cell is being used and as such, is a dynamic data representation. The left path from the node n_0 is traversed.

The traversal of the left path leads to the decision node n_1. The condition

tested at the decision node n_1 is whether destructive (a change made to the value stored or representation) or nondestructive iteration is being performed. From the parameter specification of the CONVERT algorithm it is determined that the DST parameter is passed as a variable parameter and as such, may involve changes to the values passed, therefore destructive iteration is being performed. The left path is traversed from the decision node n_1.

The traversal of the left path leads to the concatenation node n_2. The left path is traversed and a leaf is met. The leaf contains the code fragment

```
NEW(STARG)
PREVARG := NIL
```

The decision tree is traversed upward until the concatenation node n_2 is met again. The right path of the concatenation node n_2 is then traversed.

The traversal of the right path leads to the concatenation node n_3. The left path is traversed and the decision node n_4 is met. The condition tested at the decision node n_4 is whether a header cell is present or not. From the data descriptor it is determined that no header is present and the left path of the decision node n_4 is traversed. A leaf is met which contains

```
ARG := STARG
```

The decision tree is then traversed upward until the concatenation node n_3 is met again and the right path of the concatenation node n_3 is then traversed.

The traversal of the right path leads to the decision node n_5. The condition tested at this decision node n_5 is whether a size is required or present. From the data descriptor it is determined that no size is required and the right path is traversed. A leaf is met and it contains the empty statement.

The decision tree shown in Figure 10 reduces to the following:

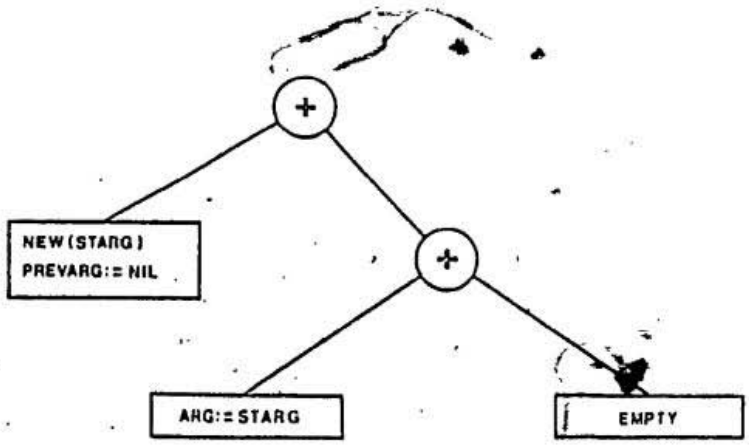


Figure 11 Reduced decision tree required by <INIT : DST>

After the concatenation process and the substitution of the string "DST" for the string "ARG", the following code fragment is generated:

<INIT : DST> ⟹

```
NEW(STDST);
PREVDST := NIL;
DST :=STDST;
```

The derivations show, by breaking the problem of code generation into one of determining required *properties (attributes)*, the code fragments can be derived in a logical and systematic manner.

7. Efficiency of Implementation

Conventional data abstraction techniques, such as procedural implementation, encapsulation, and hierarchical structuring of modules, tend to enhance several aspects of a software system: development, comprehensibility, reliability, maintainability and reusability. However, using conventional data abstraction techniques it is possible to derive inefficient implementations (poor performance characteristics).

To derive implementations with better performance characteristics involves optimization operations. There are two places in the creation of the software system that these operations can occur:

(i) At the development phase, with the selection of efficient algorithms and representations.

(ii) At the implementation phase, by making code optimizations based on equivalence preserving transformations.

The general problem of code optimizations is beyond the scope of this paper, however, the implementations generated by the event based technique must have *reasonable* time and space complexities.

Code optimization can be performed either manually or automatically [3,4,7,18,23,27]. Both techniques use a transformational system based on equivalence preserving transformations. The transformed versions of the implementation are improved relative to some measure (time or space), while preserving program correctness (points of invariance). Typical transformations which are performed are: collapsing of loops, simplifying selection, inline coding of the representation of a procedure or function , breaking encapsulation, ...

The problem domain under consideration is the automatic generation of software systems in Pascal which support the string (sequence) abstraction, using various data representations.

Let us consider the PRINT algorithm as a basis for comparison between the conventional data abstraction techniques and the event based technique presented in this paper.

The print algorithm is used to print the contents of the string (sequence).

```
PROCEDURE PRINT (<PARAM>)
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  WRITELN
  WHILE NOT (<EOF : STSRC>) DO BEGIN
    WRITE (<NOW : STSRC>)
    <MOVE : STSRC>)
  END
  WRITELN
  WRITELN
END
```

Table 8 contains a list of events in the print algorithm that need to be generated.

Table 8 Events in the print algorithm that need to be derived

Event	Description
<LOC_VAR>	local variable declaration for abstract algorithm
<INIT : SRC>	initialize source sequence for iteration
<EOF : STSRC>	test for end of source sequence
<NOW : STSRC>	produce contents of the current state of sourece sequence iteration
<MOVE : STSRC>	move to next location in the destination sequence

The data representatation used for the comparision is a singly linked list with a header cell which contains a pointer to the first element, a size and a pointer to the last element. The following type declaration is generated:

```
TYPE
  LINK = ^BODY;

  HEADER = RECORD
             SIZE : INTEGER;
             LAST : LINK;
             FIRST : LINK
           END;

  BODY = RECORD
           DATA : CHAR;  (* data field *)
           FWDPTR : LINK  (* forward pointer *)
         END;
```

If the conventional data abstraction techniques were followed (separation of implementation from use), the events (operations) should be implemented as procedure or function calls, as in the following:

```
PROCEDURE INIT (SRC : HEADER; VAR STSRC : LINK);
  BEGIN
```

```
    STSRC := SRC.FIRST
  END;

FUNCTION EOF (STSRC : LINK) : BOOLEAN;
  BEGIN
    IF STSRC = NIL THEN
      EOF := TRUE
    ELSE
      EOF := FALSE
  END;

FUNCTION NOW (STSRC : LINK) : CHAR;
  BEGIN
    NOW := STSRC^.DATA
  END;

PROCEDURE MOVE (VAR STSRC : LINK);
  BEGIN
    STSRC := STSRC^.FWDPTR
  END;

PROCEDURE PRINT (SRC : HEADER);
  VAR
    STSRC : LINK;
  BEGIN
    INIT(SRC,STSRC);
    WRITELN;
    WHILE NOT EOF(STSRC) DO BEGIN
      WRITE(NOW(STSRC));
      MOVE(STSRC)
    END;
    WRITELN;
    WRITELN
  END;
```

The implementation generated by using conventional data abstraction techniques has a large dynamic or interconnection overhead (poor time and space utilization).

If the event based technique were used, the code fragments which implement the abstract operations to be performed are derived using a deduction mechanism (decision trees) and are coded at the point of call using a technique called module expansion [4,22]. The implementation derived is the following:

```
PROCEDURE PRINT (SRC : HEADER);
 VAR
   STSRC : LINK;
 BEGIN
  STSRC := SRC.FIRST;
  WRITELN;
  WHILE NOT (STSRC = NIL) DO BEGIN
    WRITE(STSRC^.DATA);
    STSRC := STSRC^.FWDPTR
  END;
  WRITELN;
  WRITELN
 END;
```

The implementation code generated by the event based technique was found in general to run fifty percent faster than the implementation code generated by conventional data abstraction techniques. The implementation code generated by the event based technique did not have the large communication overhead associated with the code generated using the conventional data abstraction techniques.

An argument against the above statement is that in such a simple case a programmer who was implementing the print algorithm would use the same implementation. However, it must be remembered that the abstract operators are to be used throughout the software system and as such would be implemented using the function and procedure mechanisms resulting in the large dynamic or communication overhead. There are possibly more optimizations that could be performed on the implementation derived by the event based technique; however, the code generated met the operational requirements, it did not waste excessive amounts of space or time (reasonably efficient) and had all of the advantages of the code generated by the conventional techniques.

8. Implementation Examples

In the implementation phase of program deveolpment, the abstract algorithm is implemented by deriving the code fragments representing the abstract operations invoked by the events. The code fragments are then substituted inline for the event occurrences in the abstract algorithm. The result is the implementation of the abstract algorithm using a particular data representation.

This section will show the generation of the implementation code representing the abstract algorithms: CONVERT, INSERT and RELATION, using various data representations.

8.1. Convert algorithm

The convert algorithm has been introduced in previous sections as a means of showing how the event based techniques work. This section will show the generation of implementations of the convert algorithm using various data representations.

The convert algorithm is used to convert from one representation (SOURCE - SRC) to another (DESTINATION - DST). The convert algorithm is the following:

```
PROCEDURE CONVERT ( <PARAM> )
  <DOC_VAR>
BEGIN
  <INIT : SRC>
  <INIT : DST>
  WHILE NOT ( <EOF : STSRC>) DO BEGIN
    <NOW : STDST> := <NOW : STSRC>
    <MOVE : STSRC>
```

```
    <MOVE : STDST>
  END
  <CLEANUP : STSRC>
  <CLEANUP : STDST>
END
```

Table 9 contains the list of events in the convert algorithm that need their code fragments generated.

Table 9 Events in the convert algorithm that need to be derived

Event	Description
<PARAM>	parameter declaration for abstract algorithm
<LOC_VAR>	local variable declaration for abstract algorithm
<INIT : SRC>	initialize source sequence for iteration
<INIT : DST>	initialize destination sequence for iteration
<EOF : STSRC>	test for end of source sequence
<NOW : STSRC>	produce contents of current state of source sequence iteration
<NOW : STDST>	produce contents of current state of destination sequence iteration
<MOVE : STSRC>	move to next location in source sequence
<MOVE : STDST>	move to next location in destination sequence
<CLEANUP : STSRC>	perform clean up operation required for source sequence
<CLEANUP : STDST>	perform clean up operation required for destination sequence

The first example shows the conversion from a singly linked list representation with

no header cell to a doubly linked ring with a header cell that contains the following information: first (pointer to first cell), last (pointer to last cell) and a size (number of items in the ring).

The diagram shown in Figure 12 shows a simple example:

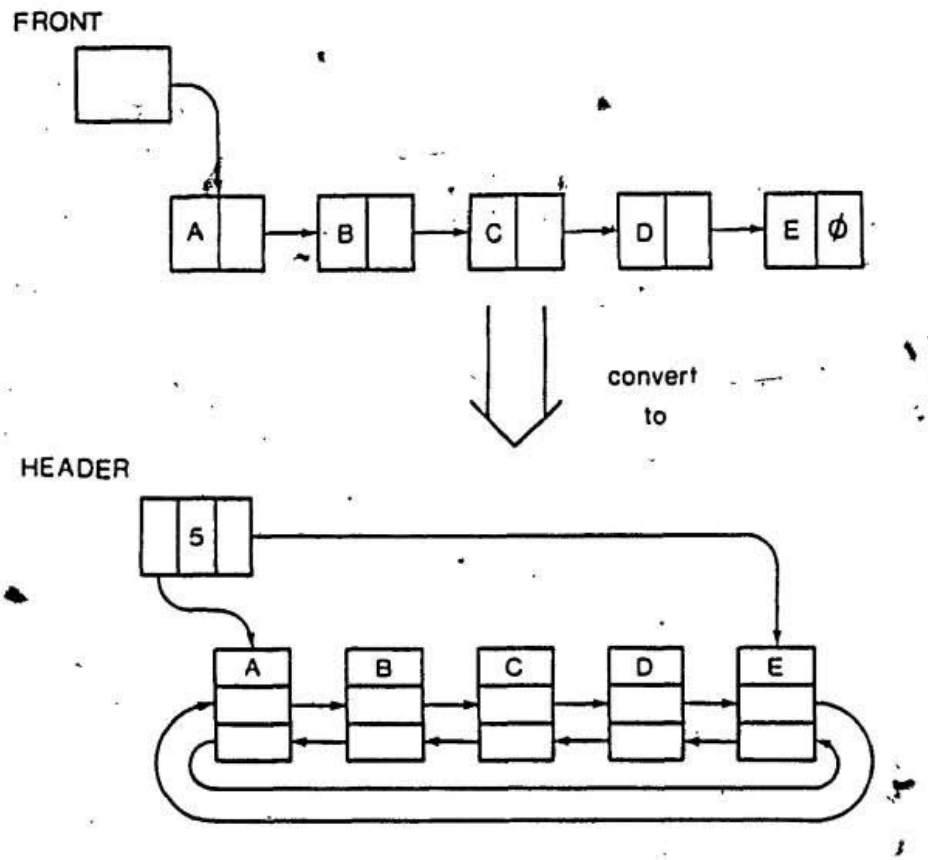


Figure 12 Conversion from a singly linked list to a doubly linked list

Where FRONT is a pointer to the first element in the singly linked list and HEADER is a record which contains: pointer to the first element, pointer to the last element and a size of the doubly linked ring.

The following type declaration must be generated:

```
TYPE

   LINK = ^BODY

   BODY = RECORD
          DATA : CHAR;  (* data field *)
          FWDPTR : LINK  (* forward pointer *)
         END;


   LINK2  = ^BODY2;

   HEADER2 = RECORD
             SIZE2 : INTEGER;  (* size field *)
             LAST2 : LINK2;    (* pointer to last *)
             FIRST2 : LINK2    (* pointer to first *)
            END;

   BODY2 = RECORD
           DATA2 : CHAR;    (* data field *)
           FWDPTR2 : LINK2; (* forward pointer *)
           BKPTR2 : LINK2   (* backward pointer *)
          END;
```

The code fragments representing the events in Table 9 would be derived in a manner described in Section 6.0. Using a decision tree structure, determine the attributes that hold for the given data representation and support these attributes by generating the code fragments to perform the required abstract operations.

The code fragments generated for the convert algorithm using a singly linked list representation (SRC) and a doubly linked ring representation (DST) are the following:

<PARAM> ⟹ SRC : LINK; VAR DST : LINK2

<LOC_VAR> ⟹
```
VAR
   I : INTEGER;
   STSRC : LINK;
   PREVDST, STDST : LINK2;
```

<INIT : SRC> ⟹ STSRC := SRC;

```
<INIT : DST>      ⟹   NEW(STDST);
                       DST.FIRST2 := STDST;
                       I := 0;
                       PREVDST := NIL;

<EOF : STSRC>     ⟹   STSRC = NIL

<NOW : STSRC>     ⟹   STSRC^.DATA

<NOW : STDST>     ⟹   STDST^.DATA2

<MOVE : STSRC>    ⟹   STSRC := STSRC^.FWDPTR;

<MOVE : STDST>    ⟹   PREVDST := STDST;
                       NEW(STDST);
                       STDST^.BKPTR2 := PREVDST;
                       I := I + 1;
                       PREVDST^.FWDPTR2 := STDST

<CLEANUP : STSRC> ⟹   (* EMPTY SEQUENCE *)

<CLEANUP : STDST> ⟹   DST.SIZE2 := I;
                       DST.LAST2 := PREVDST;
                       IF PREVDST = NIL THEN
                         DST.FIRST2 := NIL
                       ELSE BEGIN
                         PREVDST^.FWDPTR2 := DST.FIRST2;
                         DST.FIRST2^.BKPTR2 := PREVDST
                       END;
                       DISPOSE(STDST)
```

The code fragments representing an event can be quite complicated due to the data representation in use and the type of operation being performed.

The convert algorithm is transformed into the final implementation by the inline coding of the code fragments representing the events.

```
PROCEDURE CONVERT (SRC : LINK; VAR DST : HEADER2);

VAR
   I : INTEGER;
   STSRC : LINK;
   PREVDST, STDST : LINK2;
```

```
BEGIN
   STSRC := SRC;

   NEW(STDST);
   DST.FIRST2 := STDST;
   I := 0;
   PREVDST := NIL;

   WHILE NOT (STSRC = NIL) DO BEGIN

      STDST^.DATA := STSRC^.DATA;

      STSRC := STSRC^.FWDPTR;

      PREVDST := STDST;
      NEW(STDST);
      STDST^.BKPTR2 := PREVDST;
      I := I + 1;
      PREVDST^.FWDPTR2 := STDST

   END

   DST.SIZE2 := I;
   DST.LAST2 := PREVDST;
   IF PREVDST = NIL THEN
      DST.FIRST2 := NIL
   ELSE BEGIN
      PREVDST^.FWDPTR2 := DST.FIRST2;
      DST.FIRST2^.BKPTR2 := PREVDST
   END;
   DISPOSE(STDST)

END; (* CONVERT *)
```

Another example, is the conversion from a doubly linked ring with a header cell which contains a pointer to the first cell, a pointer to the last cell, and a size to an array representation with a size. The diagram shown in Figure 13 shows a simple example.

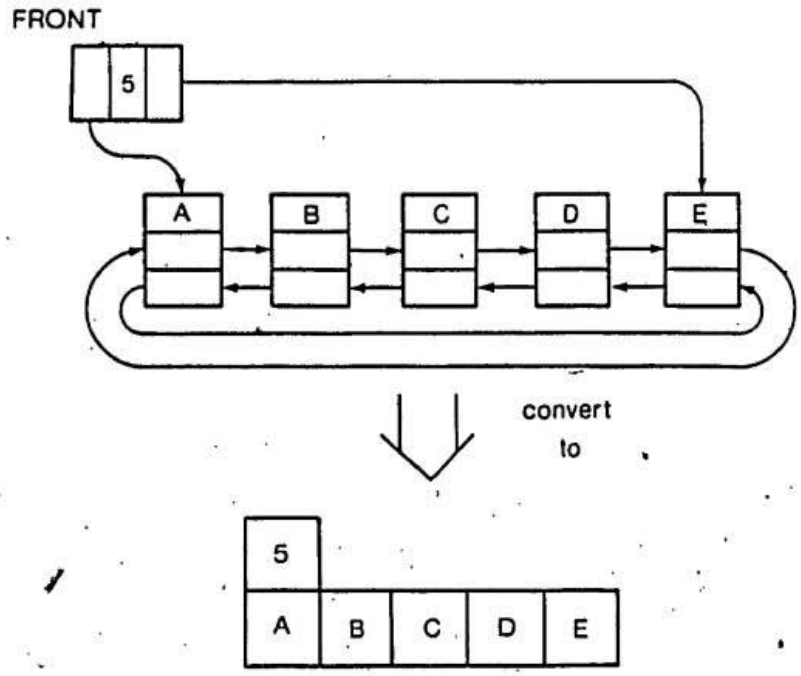


Figure 13 Conversion from a doubly linked ring to an array representation

With the following type declaration being generated:

```
TYPE
   LINK   = ^BODY;

   HEADER = RECORD
              SIZE : INTEGER;  (* size field *)
              LAST : LINK;     (* pointer to last *)
              FIRST : LINK     (* pointer to first *)
            END;

   BODY   = RECORD
              DATA : CHAR;     (* data field *)
              BKPTR : LINK;    (* backward pointer *)
              FWDPTR : LINK    (* forward pointer *)
            END;

   RECORD2 = RECORD
               NAME2 : ARRAY[1..5] OF CHAR; (* data field *)
               SIZE2 : INTEGER              (* size field *)
             END;
```

The code fragments generated for the convert algorithm using a doubly linked ring representation (SRC) and an array representation (DST) are the following:

<PARAM>	⟹	SRC : HEADER ; VAR DST : RECORD2
<LOC_VAR>	⟹	VAR STDST : INTEGER; STSRC : LINK;
<INIT : SRC>	⟹	STSRC := SRC.FIRST;
<INIT : DST>	⟹	STDST := 1;
<EOF : STSRC>	⟹	STSRC = NIL
<NOW : STSRC>	⟹	STSRC^.DATA
<NOW : STDST>	⟹	DST.NAME2[STDST]
<MOVE : STSRC>	⟹	STSRC := STSRC^.FWDPTR; IF STSRC = SRC.FIRST THEN STSRC := NIL;
<MOVE : STDST>	⟹	STDST := STDST + 1
<CLEANUP : STSRC>	⟹	(* EMPTY SEQUENCE *)
<CLEANUP : STDST>	⟹	DST.SIZE2 := SRC.SIZE

The convert algorithm is transformed by the inline coding of the code fragments representing the events:

```
PROCEDURE CONVERT (SRC : HEADER; VAR DST : RECORD2);

VAR
   STDST : INTEGER;
   STSRC : LINK;

BEGIN

   STSRC := SRC.FIRST;

   STDST := 1;

   WHILE NOT (STSRC = NIL) DO BEGIN
```

```
      DST.NAME2[STDST] := STSRC^.DATA;

      STSRC := STSRC^.FWDPTR;
      IF STSRC = SRC.FIRST THEN
         STSRC := NIL;

      STDST := STDST + 1

   END

   DST.SIZE2 := SRC.SIZE

END; (* CONVERT *)
```

The examples using the convert algorithm show, by changing the data representation, that the code fragments representing the events change dramatically. The convert algorithm implementations performs the same action, however its implementations are data representation dependent.

8.2. Insert algorithm

The insert algorithm is used to insert a string (PAT) into another string (DST) after a specified position (POS). The insert algorithm is the following:

```
PROCEDURE INSERT ( <PARAM> )
  <LOC_VAR>
BEGIN
  IFF POS <= SIZE(DST) THEN BEGIN
    <INIT : DST,POS>
    <INIT : PAT>
    WHILE NOT ( <EOF : STPAT> ) DO BEGIN
      <NOW : STDST> := <NOW : STPAT>
      <MOVE : STDST>
```

```
        <MOVE : STPAT>
      END
    ENDIFF
    <CLEANUP : STPAT>
    <CLEANUP : STDST>
  END
```

The IFF - ENDIFF notation shown in the insert algorithm, represents a conditional statement. The statement is only generated if a specified condition is true. In this case, the statement is generated if the data representation used is an array. The conditional statement is included because it is an error to try to access a component outside the bounds of an array.

Table 10 contains a list of the events in the insert algorithm that need to be generated.

Table 10 Events in the insert algorithm that need to be derived

Event	Description
<PARAM>	parameter declaration for abstract algorithm
<LOC_VAR>	local variable declaration for abstract algorithm
<INIT : PAT>	initialize pattern sequence for iteration
<INIT : DST,POS>	initialize destination sequence for iteration, initialize to a specified position
<EOF : STPAT>	test for end of pattern sequence
<NOW : STPAT>	produce contents of current state of pattern sequence iteration
<NOW : STDST>	produce contents of current state of destination sequence iteration
<MOVE : STPAT>	move to next location in pattern sequence
<MOVE : STDST>	move to next location in destination sequence
<CLEANUP : STPAT>	perform clean up operation required for pattern sequence
<CLEANUP : STDST>	perform clean up operation required for destination sequence

Consider insertion in an array representation which has a size associated with it. The following type declaration would need to be generated:

```
TYPE

   RECORD2 = RECORD
               NAME : ARRAY[1..80] OF CHAR;  (* data field *)
               SIZE : INTEGER                (* size field *)
             END
```

The code fragments generated for the insertion of a string (PAT) into another

string (DST) after a specified position using the insert algorithm and an array representation are the following:

<PARAM>	⟹	VAR DST : RECORD1; PAT : RECORD1; POS : INTEGER
<LOC_VAR>	⟹	VAR STDST,STPAT : INTEGER;
<INIT : DST,POS>	⟹	STDST := POS + 1; SHIFT(DST,STDST,SIZE(PAT),1);
<INIT : PAT>	⟹	STPAT := 1;
<EOF : STPAT>	⟹	STPAT > PAT.SIZE
<MOVE : STDST>	⟹	STDST := STDST + 1;
<MOVE : STPAT>	⟹	STPAT := STPAT + 1
<NOW : STDST>	⟹	DST.NAME[STDST]
<NOW : STPAT>	⟹	PAT.NAME[STPAT]
<CLEANUP : STPAT>	⟹	(* EMPTY STATEMENT *)
<CLEANUP : STDST>	⟹	(* EMPTY STATEMENT *)

The code fragments generated by most events should be obvious from the data representation in use and the operation being performed. The code fragment generated by the event <INIT : DST,POS> is not, and the following code is generated as the fragment representing the event:

```
STDST := POS + 1;
SHIFT(DST,STDST,SIZE(PAT),1);
```

The first statement is generated because the insertion is performed *after* a certain position. The second statement is a call to the procedure SHIFT, to shift right the elements after a specified position to make room for the insertion. A complete

specification of all abstract algorithms can be found in Appendix A and B.

The <CLEANUP : STDST> generates the empty statement. A destructive iteration is being performed and in the array representation case, this operation has been performed as part of the SHIFT algorithm. When using abstract algorithms which call other abstract algorithms, there is a need to know the attributes of the called algorithms.

The insert algorithm is transformed to the final implementation by the inline coding of the code fragments representing the events:

```
PROCEDURE INSERT (VAR DST : RECORD1; PAT : RECORD1;
                  POS : INTEGER);

VAR
  STDST,STPAT : INTEGER;

BEGIN

  IF POS <= SIZE(DST) THEN BEGIN

    STDST := POS + 1;
    SHIFT(DST, STDST, SIZE(PAT), 1);

    STPAT := 1;

    WHILE NOT (STPAT > PAT.SIZE) DO BEGIN

      DST.NAME[STDST] := PAT.NAME[STPAT];

      STDST := STDST + 1;

      STPAT := STPAT + 1

    END
  END
END; { INSERT }
```

Consider the insertion in a doubly linked list without a header cell. The following type declaration must be generated:

```
TYPE
```

```
LINK = ^BODY;

BODY = RECORD
    DATA : CHAR;  (* data field *)
    BKPTR : LINK;  (* backward pointer *)
    FWDPTR : LINK  (* forward pointer *)
  END;
```

The following form of the abstract algorithm for the insert is used:

```
PROCEDURE INSERT ( <PARAM> )
  <LOC_VAR>
BEGIN
  <INIT : DST,POS>
  <INIT : PAT>
  WHILE NOT ( <EOF : STPAT> ) DO BEGIN
    <NOW : STDST> := <NOW : STPAT>
    <MOVE : STDST>
    <MOVE : STPAT>
  END
  <CLEANUP : STPAT>
  <CLEANUP : STDST>
END
```

This version of the insert algorithm does not need conditional statements because in a dynamic structure it is possible to insert past the end of the string, and as such, there is no limit on the bounds.

The code fragments generated for the insertion of a string (PAT) into another string (DST) after a specified position using the insert algorithm and a doubly

linked list representation are the following:

<PARAM> ⟹

```
VAR DST : LINK; PAT : LINK;
    POS : LINK
```

<LOC_VAR> ⟹

```
VAR
  STDST,PREVDST,FRONTDST,BACKDST,
    STPAT : LINK;
```

<INIT : DST,POS> ⟹

```
PREVDST := POS;
NEW(STDST);
IF POS = NIL THEN BEGIN
  BACKDST := DST;
  FRONTDST := STDST
END ELSE
  BACKDST := POS^.FWDPTR;
```

<INIT : PAT> ⟹

```
STPAT := PAT;
```

<EOF : STPAT> ⟹

```
STPAT = NIL
```

<MOVE : STDST> ⟹

```
IF PREVDST <> NIL THEN BEGIN
  STDST^.BKPTR := PREVDST;
  PREVDST^.FWDPTR := STDST
END;
PREVDST := STDST;
NEW(STDST);
```

<MOVE : STPAT> ⟹

```
STPAT := STPAT^.FWDPTR
```

<NOW : STDST> ⟹

```
STDST^.DATA
```

<NOW : STPAT> ⟹

```
STPAT^.DATA
```

<CLEANUP : STPAT> ⟹

```
(* EMPTY STATEMENT *)
```

<CLEANUP : STDST> ⟹

```
DISPOSE(STDST);
IF PREVDST <> POS THEN BEGIN
  PREVDST^.FWDPTR := BACKDST;
  IF BACKDST <> NIL THEN
    BACKDST^.BKPTR := PREVDST;
  IF BACKDST = DST THEN BEGIN
    FRONTDST^.BKPTR := NIL;
    DST := FRONTDST
  END
END
```

The <CLEANUP : STDST> event needed to be generated in this case because the cleanup operation was not performed in another abstract algorithm.

The final implementation is generated by the inline coding of the code fragments.

```
PROCEDURE INSERT (VAR DST : LINK; PAT : LINK;
                  POS : LINK);

VAR
  STDST, PREVDST, FRONTDST, BACKDST,
  STPAT : LINK;

BEGIN

  PREVDST := POS;
  NEW(STDST);
  IF POS = NIL THEN BEGIN
    BACKDST := DST;
    FRONTDST := STDST
  END ELSE
    BACKDST := POS^.FWDPTR;

  STPAT := PAT;

  WHILE NOT (STPAT = NIL) DO BEGIN

    STDST^.DATA := STPAT^.DATA;

    IF PREVDST <> NIL THEN BEGIN
      STDST^.BKPTR := PREVDST;
      PREVDST^.FWDPTR := STDST
    END;
    PREVDST := STDST;
    NEW(STDST);

    STPAT := STPAT^.FWDPTR

  END;

  DISPOSE(STDST);
  IF PREVDST <> POS THEN BEGIN
    PREVDST^.FWDPTR := BACKDST;
    IF BACKDST <> NIL THEN
      BACKDST^.BKPTR := PREVDST;
```

```
      IF BACKDST = DST THEN BEGIN
        FRONTDST^.BKPTR := NIL;
        DST := FRONTDST
      END
    END

END; { INSERT }
```

The examples using the insert algorithm are different from those of the convert, in that only one data representation is used and it introduces the idea of conditional statements. As before, the algorithm performs the same action, but using different data representations, the implementations derived are significantly different.

8.3. Relation algorithm

The relation algorithm is used to return a boolean value if one string (SRC1) is Equal (EQ), Not Equal (NE), Less Than or Equal (LE), Less Than (LT), Greater Than or Equal (GE) or Greater Than (GT) to a second string (SRC2). The operation to be performed is determined by the operator (OPER) parameter The relation algorithm is the following:

```
FUNCTION RELATION ( <PARAM> ) : BOOLEAN;
  <LOC_VAR>
BEGIN
  CASE OPER OF
    EQ: BEGIN
      IFF <ACCESS_SIZE : SRC1> <> <ACCESS_SIZE : SRC2> THEN
        RELATION := FALSE
      ELSE BEGIN
        EFLAG := TRUE;
        <INIT : SRC1>
        <INIT : SRC2>
        WHILE NOT (<EOF : STSRC1>) AND NOT (<EOF : STSRC2>) AND
```

```
            EFLAG DO BEGIN
          IF <NOW : STSRC1> <> <NOW : STSRC2> THEN
            EFLAG := FALSE;
          <MOVE : STSRC1>
          <MOVE : STSRC2>
        END;
        IF (<EOF : STSRC1>) AND (<EOF : STSRC2>) THEN
          RELATION := EFLAG
        ELSE
          RELATION := FALSE
      ENDIFF
    END;

LT: BEGIN
      GFLAG := FALSE;
      LFLAG := FALSE;
      <INIT : SRC1>
      <INIT : SRC2>
      WHILE NOT (<EOF : STSRC1>) AND NOT (<EOF : STSRC2>) AND
            NOT LFLAG AND NOT GFLAG DO BEGIN
        IF <NOW : STSRC1> < <NOW : STSRC2> THEN
          LFLAG := TRUE
        ELSE IF <NOW : STSRC1> > <NOW : STSRC2> THEN
          GFLAG := TRUE;
        <MOVE : STSRC1>
        <MOVE : STSRC2>
      END;
      IF <EOF : STSRC1> THEN
        IF <EOF : STSRC2> THEN
          RELATION := LFLAG
        ELSE
          RELATION := NOT GFLAG
      ELSE
        RELATION := LFLAG
    END;

GT: BEGIN
      GFLAG := FALSE;
      LFLAG := FALSE;
      <INIT : SRC1>
      <INIT : SRC2>
      WHILE NOT (<EOF : STSRC1>) AND NOT (<EOF : STSRC2>) AND
            NOT LFLAG AND NOT GFLAG DO BEGIN
        IF <NOW : STSRC1> > <NOW : STSRC2> THEN
          GFLAG := TRUE
        ELSE IF <NOW : STSRC1> < <NOW : STSRC2> THEN
          LFLAG := TRUE;
        <MOVE : STSRC1>
```

```
        <MOVE : STSRC2>
      END;
      IF <EOF : STSRC1> THEN
        RELATION := GFLAG
      ELSE IF <EOF : STSRC2> THEN
        RELATION := NOT LFLAG
      ELSE
        RELATION := GFLAG
    END;

  NE: RELATION := NOT RELATION(EQ,SRC1,SRC2);

  LE: RELATION := NOT RELATION(GT,SRC1,SRC2);

  GE: RELATION := NOT RELATION(LT,SRC1,SRC2)

 END

END; { RELATION }
```

The IFF - ENDIFF notation in the relation algorithm , represents a conditional statement. The statement is generated only if the data representation used contains a size component.

Table 11 contains a list of the events in the relation algorithm that need their representations generated.

Table 11 Events in the relation algorithm that need to be derived

Event	Description
<PARAM>	parameter declaration for abstract algorithm
<LOC_VAR>	local variable declaration for abstract algorithm
<ACCESS_SIZE : SRC1>	access size component of src1 sequence
<ACCESS_SIZE : SRC2>	access size component of src2 sequence
<INIT : SRC1>	initialize src1 sequence for iteration
<INIT : SRC2>	initialize src2 sequence for iteration
<EOF : STSRC1>	test for end of src1 sequence
<EOF : STSRC2>	test for end of src2 sequence
<NOW : STSRC1>	produce contents of current state of src1 sequence iteration
<NOW : STSRC2>	produce contents of current state of src2 sequence iteration
<MOVE : STSRC1>	move to next location in src1 sequence
<MOVE : STSRC2>	move to next location in src2 sequence

Consider using a singly linked list representation with a header cell which contains a pointer to the first cell and a size as the data representation. The following type declaration must be generated:

```
TYPE
   LINK = ^BODY;

   HEADER = RECORD
          SIZE : INTEGER; (* size field *)
          FIRST : LINK    (* pointer to first *)
        END;

   BODY = RECORD
```

```
    DATA : CHAR; (* data field *)
    FWDPTR : LINK (* forward pointer *)
  END;
```

The code fragments generated for the relation algorithm using a singly linked list representation are the following:

<PARAM>	⟹	OPER : OP; SRC1,SRC2 : HEADER
<LOC_VAR>	⟹	VAR STSRC1,STSRC2 : LINK; EFLAG,GFLAG,LFLAG : BOOLEAN;
<ACCESS_SIZE : SRC1>	⟹	SRC1.SIZE
<ACCESS_SIZE : SRC2>	⟹	SRC2.SIZE
<INIT : SRC1>	⟹	STSRC1 := SRC1.FIRST;
<INIT : SRC2>	⟹	STSRC2 := SRC2.FIRST;
<EOF : STSRC1>	⟹	STSRC1 = NIL
<EOF : STSRC2>	⟹	STSRC2 = NIL
<NOW : STSRC1>	⟹	STSRC1^.DATA
<NOW : STSRC2>	⟹	STSRC2^.DATA
<MOVE : STSRC1>	⟹	STSRC1 := STSRC1^.FWDPTR;
<MOVE : STSRC2>	⟹	STSRC2 := STSRC2^.FWDPTR;

The relation algorithm is transformed to the final implementation by the inline coding of the code fragments.

```
FUNCTION RELATION (OPER : OP; SRC1,SRC2 : HEADER) : BOOLEAN;

VAR
  STSRC1,STSRC2 : LINK;
  EFLAG,GFLAG,LFLAG : BOOLEAN;

BEGIN
```

```
CASE OPER OF

 EQ: BEGIN
     IF SRC1.SIZE <> SRC2.SIZE THEN
       RELATION := FALSE
     ELSE BEGIN
       EFLAG := TRUE;
       STSRC1 := SRC1.FIRST;
       STSRC2 := SRC2.FIRST;
       WHILE NOT (STSRC1 = NIL) AND NOT (STSRC2 = NIL) AND
            EFLAG DO BEGIN
         IF STSRC1^.DATA <> STSRC2^.DATA THEN
           EFLAG := FALSE;
         STSRC1 := STSRC1^.FWDPTR;
         STSRC2 := STSRC2^.FWDPTR;
       END;
       IF (STSRC1 = NIL) AND (STSRC2 = NIL) THEN
         RELATION := EFLAG
       ELSE
         RELATION := FALSE
     END
   END;

 LT: BEGIN
     GFLAG := FALSE;
     LFLAG := FALSE;
     STSRC1 := SRC1.FIRST;
     STSRC2 := SRC2.FIRST;
     WHILE NOT (STSRC1 = NIL) AND NOT (STSRC2 = NIL) AND
          NOT LFLAG AND NOT GFLAG DO BEGIN
       IF STSRC1^.DATA < STSRC2^.DATA THEN
         LFLAG := TRUE
       ELSE IF STSRC1^.DATA > STSRC2^.DATA THEN
         GFLAG := TRUE;
       STSRC1 := STSRC1^.FWDPTR;
       STSRC2 := STSRC2^.FWDPTR
     END;
     IF STSRC1 = NIL THEN
       IF STSRC2 = NIL THEN
         RELATION := LFLAG
       ELSE
         RELATION := NOT GFLAG
     ELSE
       RELATION := LFLAG
   END;

 GT: BEGIN
     GFLAG := FALSE;
```

```
            LFLAG := FALSE;
            STSRC1 := SRC1.FIRST;
            STSRC2 := SRC2.FIRST;
            WHILE NOT (STSRC1 = NIL) AND NOT (STSRC2 = NIL) AND
                NOT LFLAG AND NOT GFLAG DO BEGIN
              IF STSRC1^.DATA > STSRC2^.DATA THEN
                GFLAG := TRUE
              ELSE IF STSRC1^.DATA < STSRC2^.DATA THEN
                LFLAG := TRUE;
              STSRC1 := STSRC1^.FWDPTR;
              STSRC2 := STSRC1^.FWDPTR
            END;
            IF STSRC1 = NIL THEN
              RELATION := GFLAG
            ELSE IF STSRC2 = NIL THEN
              RELATION := NOT LFLAG
            ELSE
              RELATION := GFLAG
          END;

        NE: RELATION := NOT RELATION(EQ,SRC1,SRC2);

        LE: RELATION := NOT RELATION(GT,SRC1,SRC2);

        GE: RELATION := NOT RELATION(LT,SRC1,SRC2)

      END
    END; { RELATION }
```

The implementation derived for the relation algorithm shows that it is possible to test properties of the string abstfaction by nondestructively iterating over the data representation.

The algorithms shown in this Section 8.0 and Table 4 have been implemented and tested using the data representations shown in Section 5.0 .

9. Summary

The work presented in this thesis was an investigation into the automation of the implemention of a limited data abstraction (string) in a conventional programing language (Pascal).

Conventional data abstraction techniques such as procedural implementation of the data abstraction and its set of operations, encapsulation of design decisions and hierarchical structuring of modules, at the implementation level, enhance the software development, comprehensibility, reliability, maintainability and reusability. However, with conventional data abstraction techniques, there are two problem areas: the manual implementation of the data abstraction and its set of operations, and the effficiency of the resulting implementation.

The general structure of the event based program transformation technique can be seen in Figure 1. An abstract algorithm notation is used to represent the process to be performed. The abstract algorithm contains events which are used to indicate points in the algorithm at which abstract operations are to be performed. The system, when given a description of the data representation to be used, generates the code fragments which represent the implementation code for the data abstraction operations (events). The abstract algorithm is transformed into the final program by using an event based program transformation technique, where the transformations take the form of text-for-text substitutions for the specific instance of an event and it implementation code (code fragment) in the abstract algorithm.

The technique presented in this thesis has certain advantages:

(i) The event based technique is somewhat like a macro-processing technique. However, the technique goes beyond macro-processing in that it

utilizes a knowledge based approach to code generation, since code generation is rule-based and depends on the attributes of any chosen representation. In this context, the use of conditional macro expansion can be thought as a primitive form of knowledge engineering, based on the use of "if-then" rules.

(ii) Viewing the event based program transformation technique as a primitive form of knowledge engineering:

(a) Conditional code generation is treated more explicitly as knowledge engineering.

(b) The knowledge engineering is supported by the extraction of attributes from the data representation descriptor.

(iii) The macro facility, which plants the operator implementation at the point of call, is used to overcome the large dynamic or communication overhead required when using conventional data abstraction techniques.

(iv) The ability to express an abstract algorithm and have a system generate the implementation details, at present can only be done for a fixed abstraction with a limited set of representations.

The event based program transformation technique has certain disadvantages:

(i) The abstract algorithm at present can be used to traverse the data abstraction in one direction only.

(ii) The abstract algorithm is used to specify a process to be performed. It should be written in terms of the data abstraction only, however, from the point of efficiency it was necessary in some cases to incorporate con-

dition code that was dependent on the data representation in use. The code generated by using just the data abstraction form of the abstract algorithm did not meet the time and space requirements placed on the software system. For example, in the size algorithm, if the data representation has a size associated with it, it is easier to access the size component rather than calculate a size.

There are a few steps that can be taken to generalize the event based program tansformation technique:

(i) The set of available data representations may be extended to take in other data representations, such as, group substrings in groups, ...

(ii) The abstract algorithm notation may be extended to incorporate other methods of iteration, such as backward scanning.

(iii) A knowledge engineering solution to the problem may be appropriate. Implement a Prolog solution with a language independent rule base and use language specific rules to generate programs in the languages of your choice. At present the system is implemented in Lisp and is used to generate Pascal programs.

(iv) Transformation techniques to optimize the resulting implementations, such as, collapsing of loops, optimising selection, ..., may be incorporated to develop more efficient implementations.

A typical implementation can be seen in Appendix C and Appendix D. By inspection it is obvious that many implementation decisions were made to derive the programs. The event based technique presented in this thesis is concerned with automating the implementation process and removing from the programmer the

task of providing the implementation details of commonly used data structures. The programming task should be performed at a higher level, where the programmer is concerned with developing a solution to the problem and then letting an automated process transform the solution into an implementation in a particular programming language. The programmer is removed from the problem of providing the implementation details of a solution and does not have to concern himself with the myraid of irrelevant details required to implement the solution.

10. Conclusion

Conventional data abstraction techniques aid in the algorithm design process but do nothing to directly support the program code required to provide the implementation details.

The event based technique presented in this thesis is concerned with automating the process of implementing data abstractions in a conventional programming language.

The event based technique uses an abstract algorithm notation to represent the process that must be performed. The abstract algorithm is written in terms of the data abstraction in use: it is in general, data representation independent. However, in certain cases, to derive a more efficient implementation, it is written in terms of the data representation in use.

A macro-facility is used which incorporates simple knowledge engineering to plant the abstract operators at the point of invocation.

The need for both compile-time abstract operators (parameter specification, local variable declarations, ...) and run-time abstract operators as requirements for automatic code generation; the conventional data abstraction mechanism focuses primarily on the use of run-time abstract operators even though the compile-time operators are of critical importance from the implementation point of view. This raises an important question. Could conventional data abstraction techniques be wrong; should compile-time operators be more formally supported? This is a topic for further research.

Bibliography

[1] Back R.J.R., "On Correct Refinement of Programs", Journal of Computer and System Sciences, vol. 23, no. 1, 1981, pp. 49-68.

[2] Backus J.,"Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs", Communications of the Association for Computing Machinery, vol. 21, no. 8, 1978, pp, 613-641.

[3] Barstow D.R.,"Knowledge-Based Program Construction",Elsevier North Holland,1979.

[4] Bastani F.B.,"Performance Improvement of Abstractions Through Context Dependent Transformations",IEEE Transactions on Software Engineering, vol. SE-10, no. 1, 1984, pp. 100-116.

[5] Brinch H.P.,"The Programming Language Concurrent Pascal",IEEE Transactions on Software Engineering, vol. 1, no. 2, 1975, pp. 199-207.

[6] Bishop J.M.,"Implementing Strings in Pascal",Software Practice and Experience, vol. 9, 1979, pp. 779-788.

[7] Cheatham T.E., Holloway G.H. and J.A. Townley,"Program Refinement By Transformation",5th International IEEE Conference on Software Engineering, San Diego, California,1981,pp. 430-437.

[8] Clocksin W.F. and C.S. Mellish,"Programming in Prolog",Springer-Verlag,1981.

[9] Darlington J. and R.M. Burstall,"A System Which Automatically Improves Programs", Acta Informatica, 6, 1976, pp. 41-60.

[10] Fonderaro J.K., Sklower K.L. and K. Layer,"The Franz Lisp Manual",University of California,1983.

[11] Gonnet G.H. and F.W. Tompa,"A Constructive Approach to the Design of Algorithms and Their Data Structures",Communications of the ACM, vol. 26, no. 11,1983,pp 912-920.

[12] Grogono P.,"Programming in Pascal", Addison-Wesley, 1978.

[13] Guarino L.R.,"The Evolution of Abstraction in Programming Languages", Carnegie-Mellon University,1978.

[14] Jackson M.A.,"JSP Handbook", Michael Jackson Systems Limited, 1983.

[15] Jackson M.A.,"Principles of Program Design", Academic Press, 1975.

[16] Jackson M.A.,"System Development", Prentice-Hall International, 1983.

[17] Kant E.,"On the Efficient Synthesis of Efficient Programs", Artifical Intelligence, vol. 20, no. 3, 1983, pp. 253-305.

[18] Kibler D.F., Neighbours J.M. and T.A. Standish, "Program Manipulation Via An Efficient Production System", SIGPLAN/SIGART Symposium on AI and Programming Languages, 1977, pp. 163-173.

[19] Kinnucan P.,"Computers That Think Like Experts",High Technology, Jan/1984, pp. 30-42.

[20] Knuth D.E.,"The Art of Computer Programming", Volume 1, Second Edition, Addison-Wesley Publishing Company, 1973.

[21] Leavenworth B.M. and J.E. Sammet,"An Overview of Nonprocedural Languages", Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices, vol. 9, no. 4, 1974, pp. 1-12.

[22] Liskov B., Synder A., Atkinson R. and C. Shaffert,"Abstraction Mechanisms in CLU", Communications of the Association for Computing Machinery, vol. 20, no. 8, 1977,pp. 564-576.

[23] Loveman D.B.,"Program Improvement By Source-to-Source Transformations", Journal of Association For Computing Machinery, vol. 24, no. 1, 1977, pp. 121-145.

[24] Low J.R.,"Automatic Data Structure Selection: An Example and Overview",Communications of the Association For Computing Machinery, vol. 21, no. 5, 1978, pp. 376-385.

[25] Madhavji N.H. and I.R. Wilson, "Dynamically Structured Data", Software-Practice and Expierence, vol. 11, 1981, pp. 1235-1260.

[26] Middleton A.G.,"Programmiag By Extensions : A Program Construction Technique", 3rd International Conference in Computer Science, Santiago, Chile, 1983.

[27] Middleton A.G. and R.B.B. Brake,"A Technique for Constructing Multi-linked Data Structures", 4th International Conference on Computer Science, Santiago, Chile, 1984, pp. 173-186.

[28] Parnas D.L.,"Designing Software for Ease of Extension and Contraction", 3rd International IEEE Conference on Software Engineering, Atlanta, Georgia, 1978, pp. 264-277.

[29] Sale A.,"Implementing Strings in Pascal - Again", Software Practice and Experience, vol. 9, 1979, pp. 839-841.

[30] Sale A.H.J.,"Strings and the Sequence Abstraction in Pascal", Software Practice and Experience, vol. 9, 1979, pp. 671-683.

[31] Standish T.A.,"Data Structure Techniques",Addison-Wesley Publishing Company, 1980.

[32] Wasserman A.I. and S. Gutz, "The Future of Programming", Communications of the ACM, vol. 25, no. 3, 1982, pp. 196-206.

[33] Weiser M.,"Program Slicing",5th International IEEE Conference on Software Engineering, San Diego, California, 1981, pp. 439-449.

[34] Wile D.S.,"Program Developements: Formal Explanations of Implementations", Communications of the ACM, vol. 26, no. 11, 1983, pp. 902-911.

[35] Wulf W.A., London R.L. and M. Shaw,"An Introduction to the Construction and Verification of Alphard Programs", IEEE Transactions on Software Engineering, vol. SE-2, no. 4, 1976, pp. 253-265.

Appendix A

This appendix contains a list of all the routines that make up the string processing package:

```
procedure append(var src1 : type1; src2 : type1);

  { the string src1 is extended by copying src2 onto the end of src1 }


procedure convert(src : type1; var dst : type2);

  { the contents of string src is copied to the new representaion dst. This routine is only
    generated when you wish to convert from one representation to another. }


procedure copy(src :type1; var dst : type1);

  { the contents of string src is copied to dst string }


procedure delete(var src : type1; pos : add; lngth : integer);

  { the src string is modified by the deletion of a number lngth characters after a position
    pos in string. When using arrays pos is integer and otherwise pos is a pointer. }


procedure empty(var src : type1);

  { the string src is initialized to the empty string. }


procedure extract(src : type1; pos : add; lngth : integer;
                  var dst : type1);

  { the dst will contain a copy of the charaters in src starting from pos in src and continuing
    for a length lngth. If an array representation is used, pos will contain an integer,
    otherwise it will contain a pointer. }


function find(src,pat : type1) : add;

  { the string src is searched for the occurrence of pat as a substring. The function returns
    the position of first character of such an occurence. If an array representation is used,
    the function will return an integer (0 if the pattern is not found) , othetwise it will
    return a pointer (nil if the pattern is not found). }
```

```
function indexof(src : type1; pos : integer) : pointer;

  { the function returns the address of the character at the pos location from the start. If
    the function passes the end of the string, nil is returned.  This function is only
    appropriate for linked representations. }


procedure insert(var dst : type1; pat : type1; pos : add);

  { insert the pattern pat after a given location in the dst string.Where location pos is either
    a pointer or integer depending if an array or linked representation is used. Use 0 or nil,
    as appropriate to insert in front of the string. }


procedure prcnvrt(src : type2);

  { will print contents of the converted representation src }


procedure print(src : type1);

  { will print the contents of the string representation src }


procedure putstring(var src : type1; value : char);

  { this will insert a single character value, at the end i of the string you are forming . }


function relation(oper : op; src1,src2 : type1) : boolean;

  { the function returns the boolean value if src1 is EQ,NE,LT,LE,GT,GE to src2. The operation
    being determined by the oper parameter. }


procedure shift(var src1 : type1; loc,num,dir : integer);

  { the function is appropriate for arrays, it is used to shift the remaining elements in an
    array, depending on pos a certain number num, in either a left shift dir = 0 or right shift
    dir = 1. Used in the insert and delete routines to make room for insertion or deletion }


function size(src : type1) : integer;

  { function returns an integer which is a count of the number of items in string src. }
```

Note

Not all these routines are needed for every data representation. The ones which may or may not be required are: convert,prcnvrt,indexof and shift.

Appendix B

This appendix contains the form of the abstract algorithms that make up the string processing package.

The [notation represents a conditional statement which is generated if a specified condition is true, it is equivalent to the conditional notation "iff-endiff" notation shown in Section 8.0.

The form of the abstract algorithms are the following:

```
PROCEDURE APPEND ( <PARAM> )
  <LOC_VAR>
BEGIN
  [STSRC1 := <ACCESS_LAST : SRC1>
  [STSRC1 := <ACCESS_SIZE : SRC1>
   <INIT : SRC1>
   WHILE NOT ( <EOF : STSRC1> ) DO
     <MOVE : STSRC1>
   [STSRC1 := STSRC1 -1
   [STSRC1 := PREVSRC1
   IF ( STSRC1 = <NULL_ADD> ) THEN
     COPY (SRC2,SRC1)
   ELSE BEGIN
     <INIT : SRC2>
     [<INIT : BODY1>
     WHILE NOT ( <EOF : STSRC2> ) DO BEGIN
       [<MOVE : STSRC1>
       [<NOW : STSRC1> := <NOW : STSRC2>
       [<NOW : STBODY1> := <NOW : STSRC2>
       [<MOVE : STBODY1>
       [<MOVE : STSRC2>
     END
     <CLEAN_UP>
   END
END (* APPEND *)


PROCEDURE CONVERT ( <PARAM>)
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  <INIT : DST>
```

```
    WHILE NOT ( <EOF : STSRC> ) DO BEGIN
      <NOW : STDST> := <NOW : STSRC>
      <MOVE : STSRC>
      <MOVE : STDST>
    END
    <CLEAN_UP>
  END (* CONVERT *)


  PROCEDURE COPY ( <PARAM>)
    <LOC_VAR>
  BEGIN
    <INIT : SRC>
    <INIT : DST>
    WHILE NOT ( <EOF : STSRC> ) DO BEGIN
      <NOW : STDST> := <NOW : STSRC>
      <MOVE : STSRC>
      <MOVE : STDST>
    END
    <CLEAN_UP>
  END (* COPY *)


  PROCEDURE DELETE ( <PARAM> )
  BEGIN
    IF SIZE(SRC) >= POS + LNGTH THEN
      SHIFT(SRC,POS + 1,LNGTH,0)
    <LOC_VAR>
  BEGIN
    <INIT : SRC,POS>
    I := 1
    WHILE NOT ( <EOF : STSRC> ) AND ( I <= LNGTH ) DO BEGIN
      <MOVE : STSRC>
      I := I + 1
    END
    IF I > LNGTH THEN
      <CLEAN_UP>
    ELSE BEGIN
      WRITELN
      WRITELN('*** ERROR TRIED TO DELETE PATTERN PASS END')
      WRITELN('*** OF STRING. PROCESS DELETE TERMINATED ***')
      WRITELN
    END
  END (* DELETE *)
```

```
PROCEDURE EMPTY ( <PARAM> )
BEGIN
  <CLEAN_UP>
END (* EMPTY *)


PROCEDURE EXTRACT ( <PARAM> )
  <LOC_VAR>
BEGIN
  IF SIZE(SRC) < POS + LNGTH - 1 THEN
    EMPTY(DST)
  ELSE BEGIN
    I := 1
    <INIT : SRC,POS>
    <INIT : DST>
    WHILE  NOT ( <EOF : STSRC> ) AND (
         I <= LNGTH
         )
         DO BEGIN
      <NOW : STDST> := <NOW : STSRC>
      I := I + 1
      <MOVE : STSRC>
      <MOVE : STDST>
    END
    IF I <= LNGTH THEN
      EMPTY(DST)
    ELSE
      <CLEAN_UP>
  END
END (* EXTRACT *)


FUNCTION FIND ( <PARAM> ) : <FN_TYPE>
  <LOC_VAR>
BEGIN
  LNGTHSRC := SIZE(SRC)
  LNGTHPAT := SIZE(PAT)
  IF (LNGTHSRC = 0) OR (LNGTHSRC < LNGTHPAT) THEN
    FIND := <NULL_ADD>
  ELSE BEGIN
    I := 1
    FLAG := TRUE
    WHILE FLAG DO BEGIN
      STSRC := I
      STSRC := INDEXOF(SRC,I)
      EXTRACT(SRC,STSRC,LNGTHPAT,DST)
```

```
      IF (RELATION(EQ,DST,PAT)) THEN BEGIN
        FLAG := FALSE
        FIND := STSRC
      END ELSE BEGIN
        I := I + 1
        IF LNGTHSRC - I + 1 < LNGTHPAT THEN BEGIN
          FLAG := FALSE
          FIND := <NULL_ADD>
        END
      END
    END
  END
END (* FIND *)


FUNCTION INDEXOF ( <PARAM> ) : LINK
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  LOCSRC := NIL
  I := 1
  FLAG := FALSE
  WHILE NOT ( <EOF : STSRC> ) AND NOT FLAG DO BEGIN
    IF I = POS THEN BEGIN
      FLAG := TRUE
      LOCSRC := STSRC
    END
    I := I + 1
    <MOVE : STSRC>
  END
  IF LOCSRC = NIL THEN BEGIN
    WRITELN
    WRITELN('*** ERROR TRIED TO GET ADDRESS OF POSITION')
    WRITELN('*** PASS END OF STRING')
    WRITELN('*** PROCESS INDEXOF RETURNS NIL')
    WRITELN
  END
  INDEXOF := LOCSRC
END (* INDEXOF *)


PROCEDURE INSERT ( <PARAM> )
  <LOC_VAR>
BEGIN
  IF POS <= SIZE(DST) THEN BEGIN
    <INIT : DST,POS>
```

```
    <INIT : PAT>
    WHILE NOT ( <EOF : STPAT> ) DO BEGIN
      <NOW : STDST> := <NOW : STPAT>
      <MOVE : STDST>
      <MOVE : STPAT>
    END
  END
  ELSE BEGIN
    WRITELN
    WRITELN('*** ERROR TRIED TO INSERT PATTERN PASS')
    WRITELN('*** END OF STRING')
    WRITELN('*** PROCESS INSERT TERMINATED')
    WRITELN
  END
  <CLEAN_UP>
END (* INSERT *)


PROCEDURE PRINT ( <PARAM>)
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  WRITELN
  WHILE NOT ( <EOF : STSRC> ) DO BEGIN
    WRITE(<NOW : STSRC>)
    <MOVE : STSRC>
  END
  WRITELN
  WRITELN
END (* PRINT *)


PROCEDURE PUTSTRING ( <PARAM> )
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  <INIT : SRCBODY>
  IF <EOF : STSRC> THEN
    <NOW : SRCBODY> := VALUE
    <NOW : STSRC> := VALUE
  ELSE BEGIN
    STSRC := <ACCESS_SIZE : SRC> + 1
    STSRC := <ACCESS_LAST : SRC>
    WHILE NOT ( <EOF : STSRC> ) DO
      <MOVE : STSRC>
    STSRC := PREVSRC
```

```
    <NOW : STSRC> := VALUE
    <NOW : SRCBODY> := VALUE
  END
  <CLEAN_UP>
END (* PUTSTRING *)


FUNCTION RELATION ( <PARAM> ) : BOOLEAN
  <LOC_VAR>
BEGIN
  CASE OPER OF
    EQ: BEGIN
        IF <ACCESS_SIZE : SRC1> <> <ACCESS_SIZE : SRC2> THEN
          RELATION := FALSE
        ELSE BEGIN
          EFLAG := TRUE
          <INIT : SRC1>
          <INIT : SRC2>
          WHILE NOT (<EOF : STSRC1>) AND NOT (<EOF : STSRC2>) AND
              EFLAG DO BEGIN
            IF <NOW : STSRC1> <> <NOW : STSRC2> THEN
              EFLAG := FALSE
            <MOVE : STSRC1>
            <MOVE : STSRC2>
          END
          IF (<EOF : STSRC1>) AND (<EOF : STSRC2>) THEN
            RELATION := EFLAG
          ELSE
            RELATION := FALSE
        END
      END
    LT: BEGIN
        GFLAG := FALSE
        LFLAG := FALSE
        <INIT : SRC1>
        <INIT : SRC2>
        WHILE NOT (<EOF : STSRC1>) AND NOT (<EOF : STSRC2>) AND
            NOT LFLAG AND NOT GFLAG DO BEGIN
          IF <NOW : STSRC1> < <NOW : STSRC2> THEN
            LFLAG := TRUE
          ELSE IF <NOW : STSRC1> > <NOW : STSRC2> THEN
            GFLAG := TRUE
          <MOVE : STSRC1>
          <MOVE : STSRC2>
        END
        IF <EOF : STSRC1> THEN
          IF <EOF : STSRC2> THEN
```

```
            RELATION := LFLAG
          ELSE
            RELATION := NOT GFLAG
        ELSE
          RELATION := LFLAG
      END
    GT: BEGIN
        GFLAG := FALSE
        LFLAG := FALSE
        <INIT : SRC1>
        <INIT : SRC2>
        WHILE NOT (<EOF : STSRC1>) AND NOT (<EOF : STSRC2>) AND
            NOT LFLAG AND NOT GFLAG DO BEGIN
          IF <NOW : STSRC1> > <NOW : STSRC2> THEN
            GFLAG := TRUE
          ELSE IF <NOW : STSRC1> < <NOW : STSRC2> THEN
            LFLAG := TRUE
          <MOVE : STSRC1>
          <MOVE : STSRC2>
        END
        IF <EOF : STSRC1> THEN
          RELATION := GFLAG
        ELSE IF <EOF : STSRC2> THEN
          RELATION := NOT LFLAG
        ELSE
          RELATION := GFLAG
      END
    NE: RELATION := NOT RELATION(EQ,SRC1,SRC2)
    LE: RELATION := NOT RELATION(GT,SRC1,SRC2)
    GE: RELATION := NOT RELATION(LT,SRC1,SRC2)
  END
END (* RELATION *)


PROCEDURE SHIFT ( <PARAM> )
  <LOC_VAR>
BEGIN
  LAST := <ACCESS_SIZE : SRC>
  STSRC := 1
  WHILE NOT ( <EOF : STSRC> ) DO
    STSRC := STSRC + 1
  LAST := STSRC
  IF DIR = 0 THEN BEGIN
    STSRC := LOC;
    WHILE STSRC <= LAST - NUM DO BEGIN
      <NOW : STSRC> := <NOW : (STSRC + NUM)>
      STSRC := STSRC + 1
```

```
    END
  END ELSE BEGIN
    STSRC := LAST
    WHILE STSRC >= LOC DO BEGIN
      <NOW : (STSRC + NUM)> := <NOW : STSRC>
      STSRC := STSRC - 1
    END
  END
  <CLEAN_UP>
END (* SHIFT *)


FUNCTION SIZE ( <PARAM> ) : INTEGER
BEGIN
  SIZE := <ACCESS_SIZE : SRC>
END (* SIZE *)
  <LOC_VAR>
BEGIN
  <INIT : SRC>
  I := 0
  WHILE NOT ( <EOF : STSRC> ) DO BEGIN
    <MOVE : STSRC>
    I := I + 1
  END
  SIZE := I
END (* SIZE *)
```

Appendix C

This appendix shows the running of the system in either interactive or data file modes.

To run the system, a user must call the lisp interpreter and load the system.

The following is a sample session using interactive mode:

```
% lisp
Franz Lisp, Opus 38.79
-> (include system.l)
[load system.l]
[load declaration.l]
[load misc.l]
[load subst.l]
[load general.l]
[load out.l]
[load interface.l]
[load operators/access.l]
[load operators/eof.l]
[load operators/loc_var.l]
[load operators/now.l]
[load operators/param.l]
[load operators/clean_up.l]
[load operators/init.l]
[load operators/move.l]
[load operators/op_set.l]
t
-> (interface)
Welcome

This system is used to create a string processing
package in Pascal.

The system has the ability to create a string processing
package using various internal data representations,
ie. arrays, lists and rings.

The system has two ways of interaction with the user.
The user may give a specification in the internal representation
in a file or a menu style.

Do you wish to use the menu type approach? (y or n)
y

Do you wish to have error messages output when you use the package
```

```
and make a blatant error? (y or n)
n

Which type of representation do you wish to use?
    array (array)
    singly linked list (list)
    doubly linked list (dlist)
    singly linked ring (ring)
    doubly linked ring (dring):
dlist

Do you wish to supply names for the fields of the data
representation? (y or n)
n

Do you wish to have a header cell? (y or n)
y

Do you wish to have a pointer to the last cell? (y or n)
y

Do you wish to have a size associated with the string? (y or n)
y

Do you wish to convert to a new representation? (y or n)
n

To control the names on the output files, please specify a
number to be used in forming the names.
200

    << Code generation sequence start >>

    << Code generation sequence end >>

The resulting program is in file 'output/code200.p'.

The abstract algorithms used are in file 'output/algor200'.

The code fragments used are in file 'output/oper200'.

A copy of the systems internal representation can be
found in 'data/data200.l'.

A copy of the Pascal data descriptor can be found
in 'output/descrip200'.
```

```
Have a nice day. Good Bye.


t
-> (exit)
```

To use the system in data file mode the following is a sample session:

```
% lisp
Franz Lisp, Opus 38.79
-> (include system.l)
[load system.l]
[load declaration.l]
[load misc.l]
[load subst.l]
[load general.l]
[load out.l]
[load interface.l]
[load operators/access.l]
[load operators/eof.l]
[load operators/loc_var.l]
[load operators/now.l]
[load operators/param.l]
[load operators/clean_up.l]
[load operators/init.l]
[load operators/move.l]
[load operators/op_set.l]
t
-> (interface)
Welcome

This system is used to create a string processing
package in Pascal.


The system has the ability to create a string processing
package using various internal data representations,
ie. arrays, lists and rings.

The system has two ways of interaction with the user.
The user may give a specification in the internal representation
in a file or a menu style.

Do you wish to use the menu type approach? (y or n)
n

Input the name of the file that contains the specification of
the data representation to be used:
data/data200.l
```

```
To control the names on the output files, please specify a
number to be used in forming the names.
200

<< Code generation sequence start >>

<< Code generation sequence end >>

The resulting program is in file 'output/code200.p'.

The abstract algorithms used are in file 'output/algor200'.

The code fragments used are in file 'output/oper200'.

A copy of the Pascal data descriptor can be found
in 'output/descrip200'.

Have a nice day. Good Bye.


t
-> (exit)
```

The internal representation for a doubly linked list with a pointer to last and a size is the following:

```
(setq spec
      '((structure dlist)
        (body1 (name . body)
               (data . data)
               (`forward . fwdptr)
               (`backward . bkptr))
        (header1 (name . header)
                 (`first . first)
                 (`last . last)
                 (size . size))))
(setq errflag 'nil)
```

From the field names and the previous example the meaning of the internal representation is obvious.

Appendix D

This appendix contains the listing of the program generated to implement the doubly linked list representation with a pointer to last and a size, as described in Appendix C.

The implementation is the following:

```
program prog(input, output);
type
  op = (EQ, NE, LT, LE, GT, GE);
  link = ^ body;
  header =
    record
      size: integer;
      last: link;
      first: link
    end;
  body =
    record
      data: char;
      bkptr: link;
      fwdptr: link
    end;

  function size(src: header): integer;
  begin
    size := src.size
  end; { size }

  function indexof(src: header; pos: integer): link;
  var
    i: integer;
    locsrc, stsrc: link;
    flag: boolean;
  begin
    stsrc := src.first;
    locsrc := nil;
    i := 1;
    flag := false;
    while not (stsrc = nil) and not flag do begin
      if i = pos then begin
        flag := true;
        locsrc := stsrc
      end;
      i := i + 1;
```

```
      stsrc := stsrc^.fwdptr
    end;
    indexof := locsrc
  end; { indexof }

  procedure copy(src: header; var dst: header);
  var
    stsrc, stdst, prevdst: link;
  begin
    stsrc := src.first;
    new(stdst);
    dst.first := stdst;
    prevdst := nil;
    while not (stsrc = nil) do begin
      stdst^.data := stsrc^.data;
      stsrc := stsrc^.fwdptr;
      prevdst := stdst;
      new(stdst);
      stdst^.bkptr := prevdst;
      prevdst^.fwdptr := stdst
    end;
    if prevdst = nil then begin
      dst.size := 0;
      dst.last := nil;
      dst.first := nil
    end else begin
      dst.size := src.size;
      dst.last := prevdst;
      prevdst^.fwdptr := nil;
      dst.first^.bkptr := nil
    end;
    dispose(stdst)
  end; { copy }

  function relation(oper: op; src1, src2: header): boolean;
  var
    stsrc1, stsrc2: link;
    eflag, gflag, lflag: boolean;
  begin
    case oper of
      EQ:
        begin
          if src1.size <> src2.size then
            relation := false
          else begin
            eflag := true;
            stsrc1 := src1.first;
            stsrc2 := src2.first;
```

```
            while not (stsrc1 = nil) and not
                  (stsrc2 = nil) and eflag do begin
               if stsrc1^.data <> stsrc2^.data then
                  eflag := false;
               stsrc1 := stsrc1^.fwdptr;
               stsrc2 := stsrc2^.fwdptr
            end;
            if (stsrc1 = nil) and (stsrc2 = nil) then
               relation := eflag
            else
               relation := false
         end
      end;
   LT:
      begin
         gflag := false;
         lflag := false;
         stsrc1 := src1.first;
         stsrc2 := src2.first;
         while not (stsrc1 = nil) and not
               (stsrc2 = nil) and not lflag
                 and not gflag do begin
            if stsrc1^.data < stsrc2^.data then
               lflag := true
            else if stsrc1^.data > stsrc2^.data then
               gflag := true;
            stsrc1 := stsrc1^.fwdptr;
            stsrc2 := stsrc2^.fwdptr
         end;
         if stsrc1 = nil then
            if stsrc2 = nil then
               relation := lflag
            else
               relation := not gflag
         else
            relation := lflag
      end;
   GT:
      begin
         gflag := false;
         lflag := false;
         stsrc1 := src1.first;
         stsrc2 := src2.first;
         while not (stsrc1 = nil) and not
               (stsrc2 = nil) and not lflag
                 and not gflag do begin
            if stsrc1^.data > stsrc2^.data then
               gflag := true
```

```
                else if stsrc1^.data < stsrc2^.data then
                  lflag := true;
                stsrc1 := stsrc1^.fwdptr;
                stsrc2 := stsrc2^.fwdptr
              end;
              if stsrc1 = nil then
                relation := gflag
              else if stsrc2 = nil then
                relation := not lflag
              else
                relation := gflag
            end;
          NE:
            relation := not relation(EQ, src1, src2);
          LE:
            relation := not relation(GT, src1, src2);
          GE:
            relation := not relation(LT, src1, src2)
        end
      end; { relation }

      procedure putstring(var src: header; value: char);
      var
        stsrc, srcbody: link;
      begin
        stsrc := src.first;
        new(srcbody);
        if stsrc = nil then
          srcbody^.data := value
        else begin
          stsrc := src.last;
          srcbody^.data := value
        end;
        src.size := src.size + 1;
        src.last := srcbody;
        if stsrc = nil then begin
          src.first := srcbody;
          srcbody^.bkptr := nil;
          srcbody^.fwdptr := nil
        end else begin
          stsrc^.fwdptr := srcbody;
          srcbody^.bkptr := stsrc;
          srcbody^.fwdptr := nil
        end
      end; { putstring }

      procedure empty(var src: header);
      begin
```

```
    src.size := 0;
    src.last := nil;
    src.first := nil
  end; { empty }

  procedure delete(var src: header; pos: link; lngth: integer);
  var
    i: integer;
    stsrc: link;
  begin
    if pos = nil then
      stsrc := src.first
    else
      stsrc := pos^.fwdptr;
    i := 1;
    while not (stsrc = nil) and (i <= lngth) do begin
      stsrc := stsrc^.fwdptr;
      i := i + 1
    end;
    if i > lngth then begin
      src.size := src.size - lngth;
      if stsrc = nil then
        src.last := pos
      else
        stsrc^.bkptr := pos;
      if pos = nil then
        src.first := stsrc
      else
        pos^.fwdptr := stsrc
    end
  end; { delete }

  procedure append(var src1: header; src2: header);
  var
    stsrc1, stsrc2, stbody1, frntbody1, prevbody1: link;
  begin
    stsrc1 := src1.last;
    if stsrc1 = nil then
      copy(src2, src1)
    else begin
      stsrc2 := src2.first;
      new(stbody1);
      frntbody1 := stbody1;
      prevbody1 := stbody1;
      while not (stsrc2 = nil) do begin
        stbody1^.data := stsrc2^.data;
        prevbody1 := stbody1;
        new(stbody1);
```

```
            stbody1^.bkptr := prevbody1;
            prevbody1^.fwdptr := stbody1;
            stsrc2 := stsrc2^.fwdptr
          end;
          dispose(stbody1);
          if frntbody1 <> prevbody1 then begin
            src1.size := src1.size + src2.size;
            src1.last := prevbody1;
            frntbody1^.bkptr := stsrc1;
            stsrc1^.fwdptr := frntbody1;
            prevbody1^.fwdptr := nil
          end
        end
      end; { append }

      procedure extract(src: header; pos:
              link; lngth: integer; var dst: header);
      var
        i: integer;
        stsrc, stdst, prevdst: link;
      begin
        i := 1;
        stsrc := pos;
        new(stdst);
        dst.first := stdst;
        prevdst := nil;
        while not (stsrc = nil) and (i <= lngth) do begin
          stdst^.data := stsrc^.data;
          i := i + 1;
          stsrc := stsrc^.fwdptr;
          prevdst := stdst;
          new(stdst);
          stdst^.bkptr := prevdst;
          prevdst^.fwdptr := stdst
        end;
        if i <= lngth then
          empty(dst)
        else if prevdst = nil then begin
          dst.size := 0;
          dst.last := nil;
          dst.first := nil
        end else begin
          dst.size := lngth;
          dst.last := prevdst;
          prevdst^.fwdptr := nil;
          dst.first^.bkptr := nil
        end;
        dispose(stdst)
```

```
end; { extract }

function find(src, pat: header): link;
var
  lngthsrc, lngthpat, i: integer;
  stsrc: link;
  dst: header;
  flag: boolean;
begin
  lngthsrc := size(src);
  lngthpat := size(pat);
  if (lngthsrc = 0) or (lngthsrc < lngthpat) then
    find := nil
  else begin
    i := 1;
    flag := true;
    while flag = true do begin
      stsrc := indexof(src, i);
      extract(src, stsrc, lngthpat, dst);
      if relation(EQ, dst, pat) then begin
        flag := false;
        find := stsrc
      end else begin
        i := i + 1;
        if lngthsrc - i + 1 < lngthpat then begin
          flag := false;
          find := nil
        end
      end
    end
  end
end; { find }

procedure print(src: header);
var
  stsrc: link;
begin
  stsrc := src.first;
  writeln;
  while not (stsrc = nil) do begin
    write(stsrc^.data);
    stsrc := stsrc^.fwdptr
  end;
  writeln;
  writeln
end; { print }

procedure insert(var dst: header; pat: header; pos: link);
```

```
var
  stdst, prevdst, frontdst, backdst, stpat: link;
begin
  prevdst := pos;
  new(stdst);
  if pos = nil then begin
    backdst := dst.first;
    frontdst := stdst
  end else
    backdst := pos^.fwdptr;
  stpat := pat.first;
  while not (stpat = nil) do begin
    stdst^.data := stpat^.data;
    if prevdst <> nil then begin
      stdst^.bkptr := prevdst;
      prevdst^.fwdptr := stdst
    end;
    prevdst := stdst;
    new(stdst);
    stpat := stpat^.fwdptr
  end;
  dispose(stdst);
  if prevdst <> pos then begin
    dst.size := dst.size + pat.size;
    prevdst^.fwdptr := backdst;
    if backdst <> nil then
      backdst^.bkptr := prevdst
    else
      dst.last := prevdst;
    if backdst = dst.first then begin
      frontdst^.bkptr := nil;
      dst.first := frontdst
    end
  end
end; { insert }

begin
  null
end.
```

Appendix E

This appendix contains the the same program as Appendix D but with error messages included.

The implementation generated is the following:

```
program prog(input, output);
type
   op = (EQ, NE, LT, LE, GT, GE);
   link = ^ body;
   header =
      record
         size: integer;
         last: link;
         first: link
      end;
   body =
      record
         data: char;
         bkptr: link;
         fwdptr: link
      end;

   function size(src: header): integer;
   begin
      size := src.size
   end; { size }

   function indexof(src: header; pos: integer): link;
   var
      i: integer;
      locsrc, stsrc: link;
      flag: boolean;
   begin
      stsrc := src.first;
      locsrc := nil;
      i := 1;
      flag := false;
      while not (stsrc = nil) and not flag do begin
         if i = pos then begin
            flag := true;
            locsrc := stsrc
         end;
         i := i + 1;
         stsrc := stsrc^.fwdptr
      end;
```

```
    if locsrc = nil then begin
        writeln;
        writeln('*** ERROR TRIED TO GET ADDRESS OF ');
        writeln('*** POSITION PASS END OF STRING ');
        writeln('*** PROCESS INDEX RETURNS NIL ');
        writeln
    end;
    indexof := locsrc
end; { indexof }

procedure copy(src: header; var dst: header);
var
    stsrc, stdst, prevdst: link;
begin
    stsrc := src.first;
    new(stdst);
    dst.first := stdst;
    prevdst := nil;
    while not (stsrc = nil) do begin
        stdst^.data := stsrc^.data;
        stsrc := stsrc^.fwdptr;
        prevdst := stdst;
        new(stdst);
        stdst^.bkptr := prevdst;
        prevdst^.fwdptr := stdst
    end;
    if prevdst = nil then begin
        dst.size := 0;
        dst.last := nil;
        dst.first := nil
    end else begin
        dst.size := src.size;
        dst.last := prevdst;
        prevdst^.fwdptr := nil;
        dst.first^.bkptr := nil
    end;
    dispose(stdst)
end; { copy }

function relation(oper: op; src1, src2: header): boolean;
var
    stsrc1, stsrc2: link;
    eflag, gflag, lflag: boolean;
begin
    case oper of
        EQ:
            begin
                if src1.size <> src2.size then
```

```
        relation := false
      else begin
        eflag := true;
        stsrc1 := src1.first;
        stsrc2 := src2.first;
        while not (stsrc1 = nil) and not
              (stsrc2 = nil) and eflag do begin
          if stsrc1^.data <> stsrc2^.data then
            eflag := false;
          stsrc1 := stsrc1^.fwdptr;
          stsrc2 := stsrc2^.fwdptr
        end;
        if (stsrc1 = nil) and (stsrc2 = nil) then
          relation := eflag
        else
          relation := false
      end
    end;
  LT:
    begin
      gflag := false;
      lflag := false;
      stsrc1 := src1.first;
      stsrc2 := src2.first;
      while not (stsrc1 = nil) and not
            (stsrc2 = nil) and not lflag and
              not gflag do begin
        if stsrc1^.data < stsrc2^.data then
          lflag := true
        else if stsrc1^.data > stsrc2^.data then
          gflag := true;
        stsrc1 := stsrc1^.fwdptr;
        stsrc2 := stsrc2^.fwdptr
      end;
      if stsrc1 = nil then
        if stsrc2 = nil then
          relation := lflag
        else
          relation := not gflag
      else
        relation := lflag
    end;
  GT:
    begin
      gflag := false;
      lflag := false;
      stsrc1 := src1.first;
      stsrc2 := src2.first;
```

```
            while not (stsrc1 = nil) and not
                (stsrc2 = nil) and not lflag and
                 not gflag do begin
               if stsrc1^.data > stsrc2^.data then
                  gflag := true
               else if stsrc1^.data < stsrc2^.data then
                  lflag := true;
               stsrc1 := stsrc1^.fwdptr;
               stsrc2 := stsrc2^.fwdptr
            end;
            if stsrc1 = nil then
               relation := gflag
            else if stsrc2 = nil then
               relation := not lflag
            else
               relation := gflag
         end;
      NE:
         relation := not relation(EQ, src1, src2);
      LE:
         relation := not relation(GT, src1, src2);
      GE:
         relation := not relation(LT, src1, src2)
   end
end; { relation }

procedure putstring(var src: header; value: char);
var
   stsrc, srcbody: link;
begin
   stsrc := src.first;
   new(srcbody);
   if stsrc = nil then
      srcbody^.data := value
   else begin
      stsrc := src.last;
      srcbody^.data := value
   end;
   src.size := src.size + 1;
   src.last := srcbody;
   if stsrc = nil then begin
      src.first := srcbody;
      srcbody^.bkptr := nil;
      srcbody^.fwdptr := nil
   end else begin
      stsrc^.fwdptr := srcbody;
      srcbody^.bkptr := stsrc;
      srcbody^.fwdptr := nil
```

```
    end
  end; { putstring }

  procedure empty(var src: header);
  begin
    src.size := 0;
    src.last := nil;
    src.first := nil
  end; { empty }

  procedure delete(var src: header; pos: link; lngth: integer);
  var
    i: integer;
    stsrc: link;
  begin
    if pos = nil then
      stsrc := src.first
    else
      stsrc := pos^.fwdptr;
    i := 1;
    while not (stsrc = nil) and (i <= lngth) do begin
      stsrc := stsrc^.fwdptr;
      i := i + 1
    end;
    if i > lngth then begin
      src.size := src.size - lngth;
      if stsrc = nil then
        src.last := pos
      else
        stsrc^.bkptr := pos;
      if pos = nil then
        src.first := stsrc
      else
        pos^.fwdptr := stsrc
    end else begin
      writeln;
      writeln('*** ERROR TRIED TO DELETE PATTERN PASS');
      writeln('*** END OF STRING ***');
      writeln('*** PROCESS DELETE TERMINATED ***');
      writeln
    end
  end; { delete }

  procedure append(var src1: header; src2: header);
  var
    stsrc1, stsrc2, stbody1, frntbody1, prevbody1: link;
  begin
    stsrc1 := src1.last;
```

```
    if stsrc1 = nil then
       copy(src2, src1)
    else begin
       stsrc2 := src2.first;
       new(stbody1);
       frntbody1 := stbody1;
       prevbody1 := stbody1;
       while not (stsrc2 = nil) do begin
          stbody1^.data := stsrc2^.data;
          prevbody1 := stbody1;
          new(stbody1);
          stbody1^.bkptr := prevbody1;
          prevbody1^.fwdptr := stbody1;
          stsrc2 := stsrc2^.fwdptr
       end;
       dispose(stbody1);
       if frntbody1 <> prevbody1 then begin
          src1.size := src1.size + src2.size;
          src1.last := prevbody1;
          frntbody1^.bkptr := stsrc1;
          stsrc1^.fwdptr := frntbody1;
          prevbody1^.fwdptr := nil
       end
    end
end; { append }

procedure extract(src: header; pos: link; lngth: integer;
          var dst: header);
var
    i: integer;
    stsrc, stdst, prevdst: link;
begin
    i := 1;
    stsrc := pos;
    new(stdst);
    dst.first := stdst;
    prevdst := nil;
    while not (stsrc = nil) and (i <= lngth) do begin
       stdst^.data := stsrc^.data;
       i := i + 1;
       stsrc := stsrc^.fwdptr;
       prevdst := stdst;
       new(stdst);
       stdst^.bkptr := prevdst;
       prevdst^.fwdptr := stdst
    end;
    if i <= lngth then
       empty(dst)
```

```
    else if prevdst = nil then begin
        dst.size := 0;
        dst.last := nil;
        dst.first := nil
    end else begin
        dst.size := lngth;
        dst.last := prevdst;
        prevdst^.fwdptr := nil;
        dst.first^.bkptr := nil
    end;
    dispose(stdst)
end; { extract }

function find(src, pat: header): link;
var
    lngthsrc, lngthpat, i: integer;
    stsrc: link;
    dst: header;
    flag: boolean;
begin
    lngthsrc := size(src);
    lngthpat := size(pat);
    if (lngthsrc = 0) or (lngthsrc < lngthpat) then
        find := nil
    else begin
        i := 1;
        flag := true;
        while flag = true do begin
            stsrc := indexof(src, i);
            extract(src, stsrc, lngthpat, dst);
            if relation(EQ, dst, pat) then begin
                flag := false;
                find := stsrc
            end else begin
                i := i + 1;
                if lngthsrc - i + 1 < lngthpat then begin
                    flag := false;
                    find := nil
                end
            end
        end
    end
end; { find }

procedure print(src: header);
var
    stsrc: link;
begin
```

```
    stsrc := src.first;
    writeln;
    while not (stsrc = nil) do begin
      write(stsrc^.data);
      stsrc := stsrc^.fwdptr
    end;
    writeln;
    writeln
  end; { print }

  procedure insert(var dst: header; pat: header; pos: link);
  var
    stdst, prevdst, frontdst, backdst, stpat: link;
  begin
    prevdst := pos;
    new(stdst);
    if pos = nil then begin
      backdst := dst.first;
      frontdst := stdst
    end else
      backdst := pos^.fwdptr;
    stpat := pat.first;
    while not (stpat = nil) do begin
      stdst^.data := stpat^.data;
      if prevdst <> nil then begin
        stdst^.bkptr := prevdst;
        prevdst^.fwdptr := stdst
      end;
      prevdst := stdst;
      new(stdst);
      stpat := stpat^.fwdptr
    end;
    dispose(stdst);
    if prevdst <> pos then begin
      dst.size := dst.size + pat.size;
      prevdst^.fwdptr := backdst;
      if backdst <> nil then
        backdst^.bkptr := prevdst
      else
        dst.last := prevdst;
      if backdst = dst.first then begin
        frontdst^.bkptr := nil;
        dst.first := frontdst
      end
    end
  end; { insert }

begin
```