Data-Driven Prototyping and Implementation
of File Processing Programs Using the
Data Transform Method

C.J. Lucena*
R.C.B. Martins*
D.D. Cowan**

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-84-28
September 1984

*Pontificia Universidade Catolica, Rio de Janeiro, Brazil

**Dept. of Computer Science, University of Waterloo,
Waterloo, Ontario, Canada, N2L 3G1
Data-Driven Prototyping and Implementation of File Processing Programs Using the Data Transform Method

Lucena, C.J., Martins, R.C.B.,
Pontificia Universidade Catolica,
Rio de Janeiro, Brazil

Cowan, D.D.,
University of Waterloo,
Waterloo Ontario Canada

ABSTRACT

This paper describes a substantial extension to the method proposed by Jackson whereby a program is designed from a description of its input and output data. By extending the concepts of input and output both structure-clash and backtracking problems can be integrated into the basic Jackson method. This technique is called the Data Transform Method [DTM]. The paper describes how DTM may be applied to file-processing problems and shows that a program schema results which can be used to assist with the semi-automatic production of software. Finally a specific problem involving backtracking is developed to illustrate how DTM is used in practice.

Keywords: data-driven programming, data transformations, data transform method, file processing, Jackson method, program design, program prototyping

"I do not know why, but I have never seen a machine that, however perfect in the philosopher's description, is perfect in its mechanical functioning. Whereas a peasant's billhook, which no philosopher has ever described, always functions as it should..." Umberto Eco "The Name of the Rose"

1. Introduction

The Data Transform Method [DTM] is an extension of the ideas of Jackson [1] and Warnier [2] which use the specifications of the input and output data to assist in determining the structure of a program.

Preliminary descriptions of the data transform method (DTM) together with accompanying examples have already been published in the literature. In [3] the DTM is described in conjunction with a classical programming problem, the sorting problem, to illustrate the concepts and show how to handle a problem that involves some backtracking. In [4] attention was centered on the solution of some programming examples, classified by Jackson [1] as structure-clash problems.

The goals of the previous publications have been to show how the limitations of a powerful technique such as Jackson's basic method can be overcome and how his basic method can be used as the basis for a far more general program-design approach, namely the DTM approach.

The present paper describes again the ideas associated with the DTM approach but uses a more concise formalism and examines the question of how prototypes of programs developed through DTM can be transformed into production programs. As in previous publications the
presentation is based on a typical file processing example; namely the problem of delimited strings [1].

The DTM uses the idea of applying data transformations to the problem data as they are specified at the problem statement level, and so uses the notions of problem reduction and problem decomposition as they are defined in the theory of problems [5]. The DTM transforms the space of solutions of the original problem into a space in which the problem can be decomposed into a set of sub-problems which are solvable by Jackson's basic method. The transformation steps are standard and put the problem in a canonical form that can be described as being an executable specification. The prototype constructed in this manner is a program design which is correct by construction.

The DTM extends the applicability of the Jackson basic method, since important problems that are not handled directly through the method (such as backtracking and structure clashes, can now be programmed through a uniform approach. The following results, which are stated here without proof, have been proven [6] and serve to characterize the relationship between the DTM and Jackson's basic method:

(i) The class of Jackson-solvable problems is properly included in the class of g.m.-general sequential machine) solvable problems.

(ii) The class of g.m.-solvable problems is properly included in the class of DTM-solvable problems

(iii) The class of structure-clash problems and the class of backtracking problems are properly included in the class of DTM solvable problems.

(iv) The class of pushdown-transducers solvable problems such as the recognition of $ww^R$, is included in the class of DTM-solvable problems.

It often happens that, a loss in efficiency accompanies the increase in generality. Although a DTM canonical solution is meant to be an executable specification or rapid prototype for a file processing problem it is important to stress that the specification as produced also allows for the derivation of an efficient final implementation. While the prototype can be developed with computer assistance, through a dialogue between a simple software system and the programmer, the process of generating an efficient implementation, although it can be guided in part by a series of well-defined steps, depends ultimately on the particular application under consideration. Section 4 presents the derivation of a practical implementation for the prototype developed in Section 3.

Before moving to Section 2, where the DTM approach is presented through a concise formalism, an abstract machine model of DTM is described which provides an early intuitive description of the method.

File-processing programs can be presented in an abstract manner as a program $P$ which accepts as input a file of elementary data items and produces a file of entities called a report. A picture showing one input file with each elementary item designated by a subscripted $e$ and a report designated by $l_n$ is shown in Figure 1. Even multiple input and output files can be viewed in this manner.

This abstract view of a program $P$ can be modified so that a new program $P_1$ accepts the elementary data items but its output is now a file or sequence of partial reports. The file of partial reports is created by recording as output each step in the report creation where the last partial report is the complete report or output. Such a picture is shown in Figure 2 where each partial report is designated by a subscripted $l$. It is clear that the original program could be obtained from $P_1$ by the simple step of discarding all but the last element of the output file.

Another level of abstraction is introduced into the DTM model where a program $P_2$ processes as input a stream of data called a paired-I/O-unit and produces as output a paired-I/O-unit. This abstraction is shown in Figure 3. The paired-I/O-units each contain the input and output which exists at a point in the operation of the program. Figure 3 only shows the initial input and final output and so the paired-I/O-units show a full input of elementary items and an
empty output and a full output of partial reports and an empty input. In other words, before the programs start operation the output is empty and after the program is finished the input is exhausted or empty. This pairing of input and output into paired-I/O-units provides a symmetry between input and output data structures and hence is the way the DTM avoids the problem of structure clashes. Program $P_1$ should be derivable from program $P_2$ since $P_1$ would just discard one-half of the paired-I/O-unit.

One final level of abstraction is necessary in the DTM. This new level is shown in Figure 4 where the program $P_3$ is introduced and is shown accepting a paired-I/O-unit as input and producing a sequence of paired-I/O-units as output. The method of generating the output is also shown in the Figure. Program $P_3$ is decomposed into a number of programs $P_{3_i}$ which incrementally process the paired-I/O-unit at the input and gradually transform it into the sequence of paired-I/O-units at the output.

At each stage $P_{3_i}$ the input is concatenated with the results of that stage to produce the output. This means that the input and output are both sequences containing a complete history of the computation to that point and the final output is a complete history of the computation. Since there is now a complete history of the computation present at each stage, there is no need to backtrack. It should be clear that program $P_2$ can be derived from $P_3$ by retaining only the last member of the output sequence.

The DTM as shown in the Figures is expanded in the next few sections. A detailed description of the approach and the coding of problems involving backtracking is presented.

The DTM is primarily mechanical in nature except for the construction of the components which constitute $P_3$. Hence software tools can be developed which should allow computer-aided development of software.
2. The Data Transform Method

DTM has been described in earlier versions of the present formalism [3,4]. When proposing a new programming methodology, in particular one that claims to extend the power of an existing method (Jackson's method), it is important to characterise precisely the problem-solving approach on which it is based. This can be done when we think about programs as solving problems and use concepts from the theory of problems [5].

A problem is a structure \( P = \langle D, O, q \rangle \) where the elements of \( D \) are the problem data, the elements of \( O \) are the solutions and \( q \) is a binary relation between \( D \) and \( O \) representing the problem's conditions.

A program solves a problem \( P \) if it defines a relation between \( D \) and \( O \) such that

\[
\forall d \in D \ q(d; f(d)) \quad (1)
\]

A solution space \( S \) for a problem is the set of all solutions, that is,

\[
S(P) = \{ f : D \to O \mid f \subseteq q \}.
\]

A derivation of a program using DTM requires that input \( (d \in D) \) and output \( (o \in O) \) specifications be given and a program be constructed in which equation (1) holds.

Jackson's basic method tries to determine initially a direct mapping between a data structure for \( d \) and a data structure for \( o \), where \( (d; o) \in \psi \). A difficulty occurs in solving certain classes of problems because backtracking and structure clashes arise. DTM starts by expressing the abstract notions of \( d \in D \) and \( o \in O \) instead of attempting to look for a data representations for these two entities.
The strategy for program derivation through the data transform method consists of applying the concept of problem reduction and decomposition while using Hoare's general data type construction mechanisms [7].

Program reduction and decomposition are applied in a way which will leave us with a set of Jackson-solvable problems.

The reduction of a problem \( P_0 = \langle D_0, O_0, q_0 \rangle \) to a problem \( P_1 = \langle D_1, O_1, q_1 \rangle \) is a pair of functions \( P_{01} = \{\text{ins}, \text{ret}\} \) where \( \text{ins} \) (for insert) is an unary function from \( D_0 \) to \( D_1 \), \( \text{ins} : D_0 \to D_1 \) and \( \text{ret} \) (for retrieve) is an unary function from \( O_1 \) to \( O_0 \), \( \text{ret} : O_1 \to O_0 \), such that, for all \( f_1 \in S(P_1) \), \( \text{ret} . f_1 . \text{ins} \in S(P_0) \). A sufficient condition for the pair \( \{\text{ins}, \text{ret}\} \) to be a reduction of \( P_0 \) to \( P_1 \) is that \( \text{ret} . q_1 . \text{ins} \subseteq q_0 \), where

\[
\text{ret} . q_1 . \text{ins} = \{ (d_0; o_0) \in D_0 \times O_0 \mid \exists o_1 \in O_1, o_0 = \text{ret}(o_1) \land (\text{ins}(d_0); o_1) \in q_1 \}.
\]

The first step of DTM consists of defining \( D_1 \) and \( O_1 \) as the cartesian product of \( D_0 \) and \( O_0 \), that is, \( D_1 = O_1 = D_0 \times O_0 \), and the functions \( \text{ins} \) and \( \text{ret} \) such that

\[
\forall d_0 \in D_0, \text{ins}(d_0) = (d_0; \lambda)
\]

and

\[
\forall d_1 \in D_1 \forall o_0 \in O_0, \text{ret}(d_0, o_0) = o_0
\]

where \( \lambda \) stands for a well defined element in \( O \) (usually the empty sequence). In other words the reduction through \( \text{ins} \) and \( \text{ret} \) makes use of the data type constructor cartesian-product (record) which is one of the three basic constructors proposed by Hoare [7]. The input and output data of \( P_1 \) have now, trivially, the same structure (independent of any chosen representations for \( d \) and \( o \)). Figure 5 illustrates this step.

![Figure 5](image)

The second step of DTM consists of defining a new reduction \( T_{12} = \{\text{make}, \text{last}\} \) of the problem \( P_1 = \langle D_1, O_1, q_1 \rangle \) to a problem \( P_2 = \langle D_2, O_2, q_2 \rangle \), where

\[
D_1 = O_1 = D \times O
\]
\[ D_2 = O_2 = D_1^*; \]

`make` \(D_1 \rightarrow D_1^*\), builds a unitary sequence from a given argument, and `last` \(D_1^* \rightarrow D_1\), returns the last element of a sequence. This reduction is needed to avoid the backtracking problem since the sequence mechanism provides a history of the computation.

Figure 5 now takes the form presented in Figure 6.

We now describe a standard program schema that decomposes the problem \(P_2\) into a series of smaller problems that can be solved through Jackson's basic method. The schema is illustrated in the diagram presented in Figure 7.

A Pascal-like pseudo-code description of the program schema described in Figure 7 is presented in Figure 8.

The function `update` for the class of file processing problems, is defined as
Program schema:

\[
\text{type } D_0 = \text{ seq of } O_0; \\
\quad O_0 = \text{ objects}; \\
\quad D_0 \times O_0 = \text{ record} \\
\quad \quad \quad i: D_0; \\
\quad \quad \quad r: O_0 \\
\quad \quad \quad \text{end}; \\
\quad \quad \quad (D_0 \times O_0)^* = \text{ seq of } D_0 \times O_0; \\
\]

\[
\text{var } x, d_0: D_0; \\
\quad y, o_0: O_0; \\
\]

Procedure \( P_2 \);

\[
\text{var } x_2, D_0 \times O_0)^*; \\
\]

\[
\begin{align*}
\text{begin} \\
x_2 &\leftarrow x_2; \\
\text{while not } \text{ finished}(x_3) \text{ do} \\
x_3 &\leftarrow \text{ update}(x_3); \\
y_2 &\leftarrow x_3 \\
\text{end}(P_2); \\
\end{align*}
\]

Procedure \( P_1 \);

\[
\text{var } x_2, y_2: (D_0 \times O_0)^*; \\
\]

\[
\begin{align*}
\text{begin} \\
x_2 &\leftarrow \text{ make}(x_1); \\
P_2; \\
y_1 &\leftarrow \text{ last}(y_2) \\
\text{end}(P_1); \\
\end{align*}
\]

Procedure \( P \);

\[
\text{var } x_1, y_1: D_0 \times O_0; \\
\]

\[
\begin{align*}
\text{begin} \\
x_1.i &\leftarrow x; \\
x_1.r &\leftarrow A; \\
P_1; \\
y &\leftarrow y_1.r; \\
\text{end}(P); \\
\end{align*}
\]

begin
\[
\text{begin} \\
x &\leftarrow \text{ copy}(d_0); \\
P; \\
o_0 &\leftarrow \text{ copy}(y) \\
\text{end}(\text{schema}). \\
\end{align*}
\]
\[
\text{update}(x_3) = \text{append}(x_3, \text{transform}(\text{last}(x_3)))
\]

where \text{transform} is a function from $D_0 \times O_0$ to $D_0 \times O_0$ which contributes to the solution of the problem.

The function \text{append} has the usual meaning of the operator normally associated with the type sequence, that is

\[
\text{append} : (D_0 \times O_0)^* \times (D_0 \times O_0) \rightarrow (D_0 \times O_0)^*
\]

and

\[
\text{append}(\langle l_1, \ldots, l_n \rangle, l) = \langle l_1, \ldots, l_n, l \rangle
\]

A correctness criterion for the program schema can be expressed as follows:

(i) \quad \text{update}(x_3) = \text{append}(x_3, \text{transform}(\text{last}(x_3)))

(ii) \quad \forall z_3 \in (D_0 \times O_0)^*, \text{smplr}(\text{transform}(z_3), i, x_3, l)

(iii) \text{smplr} is a well-founded relation in $D_0 \times D_0$ such that any $d_0 \in D_0$ is in a finite \text{smplr} chain starting at $A$: \text{smplr}(A, d_0, 1), \text{smplr}(d_0, d_0, 2), \ldots, \text{smplr}(d_0, d_0, n) \text{ (this is the usual criterion for a file processing program)}

(iv) \quad \text{last}(x_3), i = A \Rightarrow \text{finished}(x_3) = \text{true}.

Transform and \text{finished} must be specified so as to satisfy the previous conditions. We can now state the partial correctness conditions for the class of programs.

(v) \quad \forall z_3 \in (D_0 \times O_0)^*, \text{finished}(z_3) \Rightarrow q_2(z_3, \text{make}(0, i, A))

(vi) \quad \forall z_3 \in (D_0 \times O_0)^* q_2(z_3, \text{make}(0, i, A)) \Rightarrow q(\text{last}(z_3), r, d_0)

Intuitively, the relation \text{smplr} guarantees that in each step the \text{transform} function contributes something to the solution of the problem. The \text{smplr} relation, which is a well-founded relation, characterizes the empty element as a distinguished element that will necessarily be reached to accomplish the termination of the program.

Condition (v) guarantees that when the program stops $x_3$ is the solution of the problem for which the input is obtained from $d_0$ by the application of \text{ins} and \text{make}. Condition (vi) ensures that the reduction from the original problem $P_0$ to $P_2$ is good, i.e., that the element from $x_3$ obtained by the application of \text{ret} and \text{last} is the solution to the original problem with input $d_0$.

3. The Delimited Strings Problem

Jackson in his book "Principles of Program Design" [1] proposes the following problem (pg. 130; problem 10). A program component is to be designed which will analyse a character string, recognizing and printing two substrings, $S_1$ and $S_2$. $S_1$ is terminated by a character "@", and $S_2$ by a character "&"; the complete string is terminated by a character "%".

On entry to the component two items of input data are available: the complete string, and a pointer or subscript which points to a current location in the string. $S_1$ is defined to be the substring whose first character is the character pointed at and whose last character is the terminating "@"; $S_2$ has as its first character the character following the terminating "@" of $S_1$, and its last character is a terminating "&". Either or both strings may be empty of non-terminating characters.

It is known that the complete string is terminated by a "%" sign within 100 characters of the current location on entry: this fact can, and should, be relied on. However, it is not known that correct substrings $S_1$ and $S_2$ are present. If both are present and correct, a report should be printed in the form

GOOD STRING
$S_1=xxxxxxxxxx@$
$S_2=\text{yyyyy}$&
otherwise the string should be printed from the current location up to the terminator, in the form
BAD STRING
CHAR-001=f
CHAR-002=g
.
CHAR-nnn=%

To design a solution to the problem using DTM, we need to characterize first the abstract domains of $D_0$ and $O_0$. This can be done through the next program shown in Figure 9.

Program delimited strings;

```
type $D_0 = \text{file of char;}$
   $\quad O_0 = \text{partial-report;}$
   $\quad D_0 \times O_0 = \text{record}$
      $\quad i:D_0;$
      $\quad r:O_0$
   $\quad \text{end;}$
   $(D_0 \times O_0)^* = \text{file of } (D_0 \times O_0);$

var $x,d_0:D_0;$
   $\quad y,o_0:O_0;$

begin
   $x \leftarrow \text{copy}(d_0);$;
   firstred$\!(x,y);$;
   $o_0 \leftarrow \text{copy}(y)$
end.
```

**Figure 9**

The procedure `firstred` is responsible for the reduction $P_{01} = \{\text{ins,ret}\}$ and is listed in Figure 10.

Procedure `firstred(x:D_0; var y:O_0);`

```
var $x_1,y_1:D_0 \times O_0;$

begin
   $x_1,i \leftarrow x;$
   $x_1,r \leftarrow \Lambda;$
   secondred$\!(x_1,y_1);$;
   $y \leftarrow y_1.r$
end;
```

**Figure 10**
The assignments making use of the selectors \( i \) and \( r \) play the role of the functions \( ins \) and \( ret \). The procedure \( secondred \) is responsible for the second step towards placing a program design in a canonical form. The code which looks very much like the previous procedure is in Figure 11.

Procedure \( secondred(x_1:D_0 \times O_0; \var{y_1:D_0 \times O_0}) \);

\[ \text{var } x_2,y_2(D_0 \times O_0)^* \]

\begin{verbatim}
begin
    \( x_2 \leftarrow \text{make}(x_1) \);
    \( \text{firstdecomp}(x_2,y_2) \);
    \( y_1 \leftarrow \text{last}(y_2) \);
end;
\end{verbatim}

Figure 11

At this point we have finished the reduction phase and are able to proceed to the first step in the decomposition which will be expressed by the procedure \( \text{firstdecomp} \) in Figure 12.

Procedure \( \text{firstdecomp}(x_2:D_0 \times O_0)^*, \var{y_2:D_0 \times O_0}^* \) ;

\[ \text{var } x_3(D_0 \times O_0)^* \]

\begin{verbatim}
begin
    \( \text{while not length(last(x_3).i) = 0 do} \)
    \( x_3 \leftarrow \text{update}(x_3) \);
    \( y_2 \leftarrow x_3 \);
end;
\end{verbatim}

Figure 12

The functions \( \text{update} \) and \( \text{transform} \) in Figure 13 complete the canonical form used to express the program design. In fact, in the procedure \( \text{firstdecomp} \) we have already characterized the predicate \( \text{finished} \) as \( \text{length (last (x_3).i) = 0} \) as required in the problem statement.

The program designer can ignore the details of this method of program development. Only the notion of a DTM abstract machine needs to be kept in mind since all the steps described previously that led to the canonical form can be easily mechanized. In this case the details were left in the text for a better understanding of the underlying ideas.

The canonical form which provides the setting for the program design can already guarantee that the program will terminate as long as the procedure \( \text{process} \) guarantees that

\[ \text{length}(x_6) < \text{length}(x_5). \]

The question of producing the report for the delimited strings problem is made quite simple if we choose to center our attention on a character as the basic logical entity.

The procedure \( \text{process} \) in Figure 14 will be assigned the task of constructing the report \( y_6 \) from the input \( x_6 \). The construction of this report will depend on the state of the report.

The construction of this report will depend upon the history of the computation which will be contained in a variable called \( \text{type} \):

(i) if \( \text{type} \) has the value \("\#\"") then only characters other than @, &, and % have been read.

(ii) if \( \text{type} \) has the value \("@\"") then string \( S_1 \) is complete and \( S_2 \) has been partially read.
Function $update(x_3; (D_0 \times O_0)^*); (D_0 \times O_0)^*$;

\begin{align*}
\text{var } & y_3; (D_0 \times O_0)^*; \\
& x_4; (D_0 \times O_0);
\end{align*}

\begin{align*}
\text{begin} \\
& y_3 \leftarrow x_3; \\
& x_4 \leftarrow \text{last}(y_3); \\
& x_4 \leftarrow \text{transform}(x_4) \\
& \text{update} \leftarrow \text{append}(y_3, x_4) \\
\text{end}; \\
\text{end};
\end{align*}

Function $\text{transform}(x_4; D_0 \times O_0); D_0 \times O_0$;

\begin{align*}
\text{var } & x_5, x_6; D_0; \\
& y_5, y_6; O_0;
\end{align*}

\begin{align*}
\text{begin} \\
& x_5 \leftarrow x_4; i; \\
& y_5 \leftarrow y_4; r; \\
& \text{process}(x_5, y_5, x_6, y_6); \\
& \text{transform}.f \leftarrow x_6; \\
& \text{transform}.r \leftarrow y_6 \\
\text{end};
\end{align*}

(iii) if $type$ has the value "&" then string $S_2$ is complete.

(iv) if $type$ has the value "c" then the entire string has been read correctly.

(v) if $type$ has the value "w" then the string is incorrect. The reading and removal of a character from $x_5$ determines the state of the report and its form.

Note that the functions $\text{first}$ and $\text{tail}$ used at the beginning of the procedure $\text{process}$ have the usual meaning; $\text{first}$ removes the first element in a sequence and $\text{tail}$ returns the entire sequence without the first element.

The reader should note that if a direct mapping between the I/O data structures had been tried without previously generalizing the problem a backtracking situation would have arisen which is not Jackson-solvable. By transforming the problem using DTM a data-driven prototype has been produced because the procedure $\text{process}$ can be constructed using Jackson's basic method. We illustrate this by presenting in Figure 15, 16 and 17 the I/O diagrams and the process diagram using Jackson's notation.

Since process guarantees that the length of $x_6$ is always strictly smaller than $x_5$, the termination of the program is always assured. Since the length zero is reached in the states "c" or "w" it is easy to develop an argument that shows the total correctness of the program design.
Procedure \texttt{process}(x_5;D_0;y_5;O_0;\text{var } z_6;D; \text{var } y_6;O); \newline
\newline
\text{type } \texttt{state} = ("\#","@","&","C","w"); \newline
\text{partial-report} = \text{case } \texttt{state} \text{ of} \newline
\text{  "\#": record} \newline
\text{   "@": file of char;} \newline
\text{   "%": file of char} \newline
\text{   end;} \newline
\text{  "@": record} \newline
\text{   "@": file of char;} \newline
\text{   "&": file of char;} \newline
\text{   "%": file of char} \newline
\text{   end;} \newline
\text{  "c": record} \newline
\text{   "@": file of char;} \newline
\text{   "&": file of char} \newline
\text{   end;} \newline
\text{  "w": record} \newline
\text{   "%": file of char} \newline
\text{   end} \newline
\text{otherwise} \newline
\text{end;} \newline
\newline
\text{var } \texttt{type:state;} \newline
\text{letter:char;} \newline
\text{y_5,y_6:partial-report,} \newline
\begin{align*} \newline
\text{begin} \newline
\text{  letter} & := \texttt{first}(x_5); \newline
\text{  z_6} & := \texttt{tail}(x_5); \newline
\end{align*} \newline
\text{\{state determination\}} \newline
\begin{align*} \newline
\text{case } \texttt{letter} \text{ of} \newline
\text{  "@": if } \texttt{type} = "\#"\{\text{undefined}\} & \text{ then } \texttt{type} := "@" \newline
\text{else } \texttt{type} := "\#"; \newline
\text{  "&": if } \texttt{type} = "@" & \text{ then } \texttt{type} := "C" \{\text{correct}\} \newline
\text{else } \texttt{type} := "\#"; \newline
\text{  "%": if } \texttt{type} = "&" & \text{ then } \texttt{type} := "c" \newline
\text{else } \texttt{type} := "w"; \{\text{error}\} \newline
\text{otherwise} \newline
\text{  if } \texttt{type} \neq "@" \text{ or } "&" \text{ or } "%" \text{ then } \texttt{type} := "\#"; \newline
\end{align*} \newline
\text{end;
{report generation}

case type of
  "#" : begin
    y₆.@ := append(y₅.c,letter);
    y₆.% := append(y₆.% ,letter)
  end;

  "@" : begin
    y₆.@ := y₅.@ ;
    y₆.& := append(y₅.& ,letter );
    y₆.% := append(y₆.% ,letter )
  end;

  "c" : begin
    y₆.@ := y₅.@ ;
    y₆.& := y₅.&
  end;

  "w" : begin
    y₆.% := y₅.%
  end;
  otherwise
  end;
end;

Figure 14

4. Derivation of an Implementation from the Program Design.

It is not the goal of this paper to elaborate on the advantages of rapid data-driven prototyping. This issue has been extensively discussed in the literature, and has been one of the major motivations for the formulation of DTM. In the present section it is illustrated how a production program can be produced from a DTM design.

Initially Jackson's notation is used to outline the typical prototype produced by DTM through the steps described in the previous section.
Looking at Figure 18 it appears that the program can be substantially reduced in size by removing the overhead that was necessary to simulate a DTM abstract machine.

Since the two reductions defined by \{ins, ret\} and \{make, last\} always occur in the program schema (Figure 8), the procedures \textit{firstred} and \textit{secondred} can be combined into a single procedure called \textit{reduce}. The new version of the schema with this transformation is shown in Figure 19.

Before proceeding it is important that the reader be aware that the method is not being simplified or optimized but rather the implementation of the method for a Pascal-like machine is being developed. If, for instance, a fundamentally different base-machine was available, say a functional machine, the method would still hold but implementation procedures would look quite different.

In file-processing applications the history of the computation is often not required since the program is usually handling the last component of the sequence. Since after testing a data-driven prototype for a file-processing application we normally cease to be interested in the sequence of
computation produced by the executable specification, we note that the procedures \textit{firstdecomp} and \textit{update} may be merged and that the type \((D \times O)^e\) can be eliminated. These modifications lead to the program schema in Figure~20.

Finally the invocation of the procedures \textit{update} and \textit{reduce} can be replaced with the body of the procedures in the main program body. The program schema now takes the form shown in Figure~21.

Using this latest version of the program schema and replacing the general names of the variables by the names of variables used in the solution of the problem of delimited strings presented in Section~3, the schema is as shown in Figure~22.
Program schema;

\[
\text{type } D_0 = \\
\quad O_0 = \\
\quad (D_0 \times O_0) = \text{record} \\
\quad i : D_0; \\
\quad r : O_0 \\
\end{array}
\]
\[
\text{end} \\
\]
\[
(D_0 \times O_0)^* = \text{seq of } (D_0 \times O_0)
\]

\[
\text{var } d_0, x : D_0; \\
\quad o_0, y : O_0; \\
\quad x_4, z_1, y_1 : D_0 \times O_0; \\
\quad y_3, z_2, y_2, z_3 : (D_0 \times O_0)^*;
\]

\[
\text{procedure } \text{update}(D \times O)^*; \\
\quad \text{begin} \\
\quad \quad y_3 \leftarrow x_3; \\
\quad \quad y_4 \leftarrow \text{last}(x_3); \\
\quad \quad z_4 \leftarrow \text{transform}(x_4); \\
\quad \quad \text{update} \leftarrow \text{append}(y_3, z_4) \\
\quad \text{end } \{\text{update}\};
\]

\[
\text{procedure } \text{firstdecomp}; \\
\quad \text{begin} \\
\quad \quad z_3 \leftarrow z_2; \\
\quad \quad \text{while not length(last}(z_3).i) = 0 \text{ do} \\
\quad \quad \quad z_3 \leftarrow \text{update}(z_3); \\
\quad \quad y_2 \leftarrow z_3 \\
\quad \text{end } \{\text{firstdecomp}\};
\]

\[
\text{procedure } \text{reduce}; \\
\quad \text{begin} \\
\quad \quad x_1.i \leftarrow x; \\
\quad \quad x_1.r \leftarrow A; \\
\quad \quad y_2 \leftarrow \text{make}(x_1); \\
\quad \quad \text{firstdecomp}; \\
\quad \quad y_1 \leftarrow \text{last}(y_2); \\
\quad \quad y \leftarrow y_1.r \\
\quad \text{end } \{\text{reduce}\};
\]

\[
\text{begin} \\
\quad z \leftarrow \text{copy}(d_0); \\
\quad \text{reduce}; \\
\quad o_0 \leftarrow \text{copy}(y) \\
\quad \text{end } \{\text{schema}\}
\]
Program schema;

    type $D_0 =$
            $O_0 =$
               $(D_0 \times O_0) =$ record
                $i:D_0$;
                $r:O_0$
            end;

    var $d_0,x:D_0$;
        $o_0,y:O_0$;
        $z_3,z_2,y_2:(D_0 \times O_0)$;

    procedure update;

    begin
        $x_3 \leftarrow x_2$;
        while not $\text{length}(x_3,i) = 0$ do
            $x_3 \leftarrow \text{transform}(x_3)$;
            $y_2 \leftarrow z_3$
        end{update};

    procedure reduce;

    begin
        $x_2.i \leftarrow x$;
        $x_2.r \leftarrow A$;
        update;
        $y \leftarrow y_2.r$
    end{reduce}

begin
    $x \leftarrow \text{copy}(d_0)$;
    reduce;
    $o_0 \leftarrow \text{copy}(y)$
end{schema}

Figure 20
Program schema;

\[
\text{type } D_0 = \\
O_0 = \\
(D_0 \times O_0) = \text{record} \\
i : D_0; \\
r : O_0 \\
\text{end;}
\]

var \(d_0, x : D_0;\) \\
\(o_0, y : O_0;\) \\
\(z_2 : D_0 \times O_0;\)

begin \\
\(x \leftarrow \text{copy}(d_0);\) \\
\(z_2.i \leftarrow x;\) \\
\(z_2.r \leftarrow A;\) \\
while not \(\text{length}(z_2.i) = 0\) do \\
\(z_2 \leftarrow \text{update}(z_2);\) \\
\(y \leftarrow z_2.r;\) \\
\(o_0 \leftarrow \text{copy}(y);\) \\
end \{ \text{schema} \};

Figure 21
Program delimitedstrings;

type
  state = ("#","@","c","w");
  partial-report = case state of
    "#": record
      "@": file of char;
      "%": file of char
    end;
    "@": record
      "@": file of char;
      "&": file of char;
      "%": file of char
    end;
    "c": record
      "@": file of char;
      "&": file of char
    end;
    "w": record
      "%": file of char
    end;
  otherwise
  end;

D_0 = file of char;
O_0 = partial-report;
D_0 × O_0 = record
  i:D_0;
  r:O_0
end;

var
d_0,x:D_0;
o_0,y:O_0;
x_2:D_0 × O_0;

Procedure process(var x_2.i:D_0, var x_2.r:O_0);

var
  type:char;
  letter:char;

begin
  letter := first(x_2.i);
  x_2.i := tail(x_2.i);
  type := "#";
  case letter of
    "@": if type = "#" then type := "@
    else type := "#";
    "&": if type = "@" then type := "c"
    else type := "w";
    "%": if type = "&"
  end;
end;
then type := "e"
else type := "w"
otherwise
if type ≠ "@" or "&" or "%"
then type := "#"
end

case type of
"#": begin
z2.r.% := append(z2.r.@,letter);
z2.r.% := append(z2.r.% ,letter)
end;

"@": begin
z2.r.@ := z2.r.@;
z2.r.& := append(z2.r.& ,letter);
z2.r.% := append(z2.r.% ,letter)
end;

"e": begin
z2.r.@ := z2.r.@;
z2.r. & := z2.r. &
end;

"w": begin
z2.r.% := z2.r.%
end;
otherwise
end
end;

z := copy(d0);
z2.i := z;
z2.r := A;
while not length(z2.i) = 0 do
process(z2.i,z2.r);
y := z2.r;
o0 := copy(y)
end {delimitedstrings}

Figure 22
5. Conclusions

The paper has shown that a prototype program can be derived using the Jackson approach embedded in the Data Transform Method (DTM). Since most of the prototype can be characterized by a program schema it is only necessary to concentrate on the development of the lower level procedure called "process". We believe that this method of program formulation and prototyping simplifies program development. There is also an extra benefit in that total program correctness is almost a by-product.

Since there is one program schema which can be followed for every file-processing problem it should be clear that the program development process can be partially automated. In fact several prototyping tools have already been created and tested. Future work will concentrate on developing a user-friendly interface based on expert systems.

6. Acknowledgements

The work described in this paper was sponsored by several agencies and the authors gratefully acknowledge the assistance of NSERC, WATFAC, and FINEP.

7. References


