1984

Quedit

David Betts
University of Wollongong, uow@betts.edu.au

Trevor Chorvat
University of Wollongong, uow_chorvatt@uow.edu.au

John Constable
University of Wollongong

Michael Fazzolare
University of Wollongong

Craig Green
University of Wollongong

Recommended Citation
Betts, David; Chorvat, Trevor; Constable, John; Fazzolare, Michael; and Green, Craig, Quedit, Department of Computing Science, University of Wollongong, Working Paper 84-2, 1984, 58p.
http://ro.uow.edu.au/compsciwp/50

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
QUEDIT

David Betts
Trevor Chorvat
John Constable
Michael Fazzolare
Craig Green

Preprint No.84-2 March 7, 1984

Submitted in partial fulfillment of the requirements of CSCI 321 Software Project at the University of Wollongong in 1983

P.O. Box 1144, WOLLONGONG, N.S.W. 2500, Australia
Tel: (042) 270 859
Telex: 29022
PART I

TECHNICAL REPORT
This document is a technical report on the design of an initial implementation of an INTERACTIVE TEXT PROCESSING LANGUAGE (ITPL). The definition of the terms used in the report to describe the language and the corresponding editor produced can be found in the specification [i].
TABLE OF CONTENTS.

1. INTRODUCTION
   1.1 ITPL the Meta-Space 3
   1.2 Spaces and Dimensions of ITPL 3

2. LANGUAGE CONTROL
   2.1 Introduction 4
   2.2 User Command Set 4
   2.3 User Command Maintenance Set 5
   2.4 Language Implementation 6
   2.4.1 Lexical Analysis 7
   2.4.2 Parsing 8
   2.4.3 Execution 12

3. TEXT CONTROL
   3.1 Introduction 15
   3.2 Text Structure 15
   3.3 Implementation of Primitive Operations 18

4. SCREEN CONTROL
   4.1 Introduction 21
   4.2 Implementation 22

APPENDIX A
   DEFINITIONS AND TERMINOLOGY 24

APPENDIX B
   GRAMMAR OF ITPL 27

REFERENCES 29

November 15, 1983
1. INTRODUCTION

1.1 ITPL the Meta-Space

Using the terms and meanings of REINFELDS [ii] this article will attempt to illustrate the design of QUEDIT. Adopting the approach of REINFELDS [ii] we have the editing problem represented as follows.

\[
\text{editing task} \implies \text{user interface:} \\
\text{computer interface:} \\
\text{given equipment and specifications:}
\]

The language is a meta-space in that it contains the three spaces of user interface, computer interface and the equipment and specifications. The user of the language can determine his own user interface when editing without worrying about the computer interface.

1.2 The Spaces and Dimensions of ITPL

The spaces of user interface and computer interface contain the dimensions of language control, screen control and text control. This article then, contains the explanation of the dimensions of:

LANGUAGE CONTROL
TEXT CONTROL
SCREEN CONTROL

Each of the dimensions has associated with it, its user interface and computer interface. However, as the user interface aspect of screen control and text control are contained within the user interface of language control, they are to be found in the user interface section of language control.

November 15, 1983
2. LANGUAGE CONTROL

2.1 Introduction

The editor does not have a static command set but rather a dynamic (changeable) set of commands. The user space therefore allows the user to perform editing functions but has the added complication that in order to have a dynamic set of commands we have supplied a mechanism to maintain that set.

The space of computer interface has to then come to grips with the mechanism through which the language will be implemented on the computer.

This leads us to our user space containing the dimensions of:

- > User Command Set

- > User Command Maintenance Set

and the space of computer interface contains the dimension of

- > Language Implementation.

2.2 User Command Set

The dimension of the user command set has two dimensions in it. They are:

[A] Standard Library

[B] User Library

The types of commands that are available in the Standard Library are found in the definition section. The types of commands that are available in the User Library is up to each particular user. Both libraries contain macros built up from the language operators.

These operators are:

(a) INSERT - an operator that inserts a string into a piece of text.
(b) DELETE - an operator that deletes a string from a piece of text.

(c) MOVE - an operator that transfers the current working (or active position) in a piece of text to a new active position.

(d) MARK - an operator that sets the mark of the current text to the current active position.

(e) GETMARK - an operator that returns (via an integer variable) the current marked position of the current text.

(f) GETACTIVE - an operator that returns (via an integer variable) the current active position of the current text.

(g) GETEND - an operator that returns (via an integer variable) the end position of a piece of text.

(h) SEARCH - an operator that searches for a string in a piece of text.

2.3 User Command Maintenance Set

The dimension of User Command Maintenance Set has to allow the user to create, update and delete from their own user library.

The following operators are supplied:

(a) COMPILE - an operator that takes a piece of macro text and produces the appropriate executable code.

(b) EXEC - an operator that executes a piece of compiled macro code.

(c) ASSOC - an operator that places a macro into a table (i.e. ASSOCIATION TABLE) so that the macro can be referenced without accessing the SYMBOL TABLE. (the SYMBOL and ASSOCIATION tables are defined in the definition section)

(d) BIND - an operator that associates a particular (sequence) of keystrokes with a macro.

November 15, 1983
2.4 Language Implementation.

On the computer interface side of the editor the dimension of language implementation has the job of actually implementing the language (ITPL).

The overall strategy of the implementation of ITPL is to convert some input macro text into actions that change text. ITPL has the normal implementation phases of lexical analysis, parsing and execution.

Let us explain the meaning of the phases of ITPL.

The input macro code is passed to the lexical analyser as defined in Step [1] The Lexical Analyser passes its tokens to the Parser as defined in Step [2] which matches the tokens with its production rules and builds a parse tree for the macro. In the compile stage of the process variable names are entered into the SYMBOL TABLE so that static (compile time) error checking can be accomplished. After the parse tree has been built and verified the symbols are deleted from the table so as to keep the size of the table limited. There is only one SYMBOL TABLE and so all variables that are in the table at a particular stage are global to the macro that is executing. The only variable that is not deleted at the completion of the parse is the macro name which remains in the SYMBOL TABLE for the life of an edit session.

In the Executon phase (see Step [3]) the variables are re-entered in the SYMBOL TABLE at the start of the execution and redeleted at the end of the execution. The end of an execution is after the last statement has been executed.

It is advisable here for the reader to make a mental note of the fact that ITPL contains a statement compile, that does the actual compilation of a piece of macro text.

In order to achieve this goal the following mechanisms are needed:

Step [1] Lexical Analyser.


November 15, 1983
2.4.1 Lexical Analyser.

The mechanism of lexical analysis has the task of transforming input strings in the language into tokens that can be recognized by the compiler.

The Lexical Analyser for the language ITPL is constructed from input to a UNIX utilities program LEX \[iv\]. LEX recognises a set of regular expressions and performs the corresponding actions. The action in all cases is to return to the parser the token and the string associated with that token.

A token is just a unique number (integer) associated with an input string.

The tokens that the Lexical Analyser must identify are:

- an identifier - a letter (upper or lower case) followed by an arbitrary number of letters or digits or the underscore (_ ) character.
- a number - an arbitrary number of digits.
- a comment - any legal character between a `{` and a `}`.
- a string - any legal character between two single quotes. If control characters are needed in a string then the following escape sequences are allowed: \#[A - ] which are mapped into their control characters. To get a `#` in a string use `##` and to get a `"` in a string use `"`.
- a legal - any printable character.
- an illegal - any non-legal ASCII character.

So the mechanism of lexical analysis defines the interface between the user and the ITPL. On the computer level this is handled by returning tokens to the parser.

November 15, 1983
The grammar of ITPL can be found in the APPENDIX B.

The parsing mechanism of the dimension of Language Implementation is responsible for transforming a stream of tokens into some form of code that can be executed. The parser for the language ITPL is constructed from input to a UNIX utilities program YACC [iv]. YACC takes as input the stream of tokens from LEX and matches them with production rules for its source code. For each rule there may be a corresponding action and the actions in this case are to build nodes of a parse tree that is to be executed later by the Executer.

The parse tree of ITPL is a tree whose nodes correspond to a piece of macro text. The macro text consists of a macro name, its declaration list and its statement list.

Take for example the parse tree for a simple program

```
procedure show

   { an example macro to show what the parse tree }
   { for a simple macro looks like }

   text : current ;
   integer : i;

   i := 0
   if i = 0 then
      display[current]
   endif
   insert["hello world",current]
   display[current]
endproc
```

November 15, 1983
A macro name node contains the macro name as a string of characters.

A declaration list is a list of declarations where each declaration can be either a variable declaration or another macro declaration. A variable declaration is a type identifier followed by a list of identifiers of that type. The possible types are:

- integer
- string
- boolean
- text
- file

November 15, 1983
An identifier is as defined in the lexical analysis section with the added restriction the variables should be unique.

A statement list is a list of statements where each statement can be one of the following statements:

* An assignment statement is made up of a variable and an expression. The value of the expression is assigned to the variable.

* A while statement which is some expression that represents the loop guard/terminator and a list of statements to be executed as the body.

* An if statement which is an expression to be tested and a list of statements to be executed if the test is successful.

* A primitive statement consists of a primitive name and an expression list. The expression list is just a list of expressions. These expressions are evaluated by the expression evaluator and there values are returned to the primitive function, that simply uses the values as a means of knowing what changes are to be made to the text. The primitive statements are passed expression lists rather than just strings for two reasons. Firstly they need more information than just strings and secondly it is desirable to pass parameters of the form: "parameter operator parameter", to the primitive functions. The primitive statement then simply sets up a node consisting of a primitive name and a list of expressions to work on. The names and the expected expression lists are:

  ASSOC - expects an expression representing the string that is to be vectored into the association table.

  BIND - expects an expression representing the string that is to be bound and an expression representing the macro that the string is bound to.

  SYSTEM - expects an empty expression list.

  COMPILE - expects an expression representing the macro text that is to be compiled.

  EXEC - expects an expression representing the macro that is to be executed. (see step [3]).

  GETCHAR - expects an expression representing the source of the character input that is received from the terminal.

  GETDATA - expects an expression

  DISPLAY - expects an expression representing the piece of text that is to be displayed.

  INSERT - expects an expression representing the string that is to be inserted and an expression representing the text, the string is to be inserted into.

November 15, 1983
DELETE - expects an expression that represents the MARKED text that is to be deleted.

SEARCH - expects an expression that tells which direction to search, an expression that represents the string to be searched for and an expression that represents the text that is to be searched.

GETACTIVE - expects an expression that represents the text to get the active position from and an expression representing the integer that the active position is to be placed in.

GETMARK - expects an expression that represents the text to get the marked position from, and an expression representing the integer that the marked position is to be placed in.

GETEND - expects an expression that represents the text variable to get the end marker from, and an expression representing the integer that the end position will be put into.

SUBSTRING - expects an expression that represents the character in the string that you wish to start at, an expression representing the number of characters you wish extracted, an expression representing the source string (the one you want to get it from) and an expression representing the target string (the one you want to put it into).

MOVE - see Text Control.

An expression has an operator type. (i.e. plus, minus, times, div, mod, and, or, less, grt, lequal, gequal, equal, neg, concat, union, qstring & identifier.) To see what operators are defined on what types read the specification [1].

The expression also contains the type of the data being operated on (i.e. string, boolean, integer, text or file).

Depending upon the type of operation the expression record (which is part of the parse tree and simply captures the meaning of an expression) contains either:

(1) two subtrees containing the operands of a binary operator.

(2) a single subtree containing the operand of some unary operator.

(3) a number.

(4) a string.

(5) an identifier.

November 15, 1983
2.4.3 Executer

The Executer is concerned with the actual execution of a parse tree. It simply traverses the parse tree in an inorder fashion and executes each node according to the following rules:

In its initial invocation the executer takes the data descriptor that it received and checks that the macro exists (i.e. in SYMBOL TABLE) and if so calls the functions "dodeclist" and "dostatlist" on the appropriate nodes of the tree.

"Dodeclist" takes the node of the parse tree that contains the declaration list and for each variable declaration places an entry into the SYMBOL TABLE after checking that the variable does not already exist. For each macro that is declared enters its name and variables in the SYMBOL TABLE.

"Dostatlist" takes the statement list and for each statement calls one of the following functions:

(a) doassign - sets a data descriptor to the result of the expression (see "doexpr") and copies the contents of this data descriptor designated by the identifier string.

(b) dowhile - sets a temporary data descriptor to the value of the while loop guard and as long as the guard's boolean value is not zero it executes (via "dostatlist") the statements in the while loop.

(c) doif - sets a temporary data descriptor to the value of the if expression guard and if that data descriptor's boolean value is not equal to zero the statement list is executed.

(d) doprimitive - calls the appropriate primitive function to execute the primitive.

The functions are:

  doassociate :- (e.g. if a string is contained in an expression list and it is passed to "doassociate" then this string is searched for in the ASSOCIATION TABLE and a temporary data descriptor is made from the expression by calling entry in the ASSOCIATION TABLE and returns the corresponding data descriptor.

  dobind :- takes its input expression list and converts it into two temporary data descriptors that contain the string to be bound and the macro that it is to be bound to.

  dogetch :- takes its input expression list and converts it into a temporary data descriptor and calls the function "getchar" which assigns the result of the getchar
to the string value of the data descriptor.

dodisplay :- takes it's input expression list and converts it into a data descriptor and passes the data descriptor to "display" which then handles the displaying (see Screen Control).

docompile :- takes it's input expression list and converts it into two data descriptors that describe the place where the macro is to be obtained and the name of the program as stored in the SYMBOL TABLE.

dosysshell :- calls SHELL which puts the user in another UNIX shell.

doinsert :- calls Insert (see Text Control) with the appropriate data descriptors.

dodelete :- calls Delete (see Text Control) with the appropriate data descriptors.

domove :- calls MoveActive (see Text Control) with the appropriate data descriptors.

domark :- calls Mark (see Text Control) with the appropriate data descriptors.

dogetmark :- calls GetMark (see Text Control) with the appropriate data descriptors.

dogetactive :- calls GetActive (see Text Control) with the appropriate data descriptors.

dogetend :- calls GetEnd (see Text Control) with the appropriate data descriptors.

dosubstring :- calls the substring (see Text Control) with the appropriate data descriptors.

The appropriate data descriptors are set up by the function doexpr. Do expression is a recursive function that calculates the value of an expression. An expression is either of the form of a binary operator applied to two binary operands or an unary operator applied to a single operand. All expressions reduce to a data descriptor that contains the semantics of the result of the expression. This data descriptor contains either the value of an identifier, the value of a string, the value of an integer, or a true or false value for a boolean expression.

Therefore doexpr simply calls itself recursively on an expressions subexpressions until it achieves completion of the whole expression. As a simple example consider.
a *( b + c)  
doexpr is called on the whole expression originally  
it then calls itself on the subexpression ( b + c )  
the result of this is passed back to the first call  
then finally  
a * result ( doexpr ( b + c ) ) is computed and passed back
3. TEXT CONTROL

3.1 Introduction

The descriptions in this section assume some familiarity with the implementation of QUEDIT. (i.e. structure and use of data descriptors (dd’s)).

The main ideas to note are as follows:

* the editor can load any file into its internal text variables.
* all editing occurs on the internal text variables.
* the copy of the original text file is loaded into a virtual array as a continuous stream of ASCII characters. There is no ‘stripping’ of control characters. Control characters imbedded in text are displayed in a distinguishable visual format (inverse-video mode).
* text is then said to consist of two strings of characters with an implied active position between the two strings. A string can be a null string. i.e.

\[
text = \langle \text{string1} \rangle \langle \text{string2} \rangle
\]

| implied active position

3.1 Text Structure

The main structure used by QUEDIT to process the text consists of:

(i) 3 integer indexes: active - implied active position.

\[
\text{mark} \quad \text{marked position.}
\]
\[
\text{rear} \quad \text{indicates how many characters the text contains.}
\]

(ii) A virtual array of characters.

The virtual array of characters is implemented as a doubly linked list of index blocks. Each index block contains an array of text block descriptors, each of which describes where a text block can be found and how many characters it contains. The text blocks are of a uniform size (512 bytes) and correspond to disk blocks contained in the

November 15, 1983
edit program temporary file. The text blocks are identified by their disk block number.

The only operations which require direct access to the text structure are:

(i) insert
(ii) delete
(iii) readtextchar

Insert and delete must alter the contents of the text structure while readtextchar supplies a uniform and text implementation independent interface to operations which read the text such as search and display.

Insert, delete and readtextchar require the relevant text block to be in memory in order to access or alter it. The text buffer resides in memory and consists of a small number of text frames which are the same size as the text blocks. In order to access or alter text contained in the text blocks it is necessary that the text block be read into a text frame. Text blocks are updated by writing from a text frame to the relevant text block. Of course, the usual practices of deferred write and write modified only apply.

The operations of insert, delete and readtextchar will have the following effects on the text frames, the text blocks and the index blocks. The index blocks contain referencing information (block numbers) about the text blocks so that a virtual sequence of characters is maintained. (Define the active frame as the text frame containing the active position.)

(a) Insertion into the active frame.
   If the string to be inserted at the active position will fit into the active frame then the characters after the active position are rippled up to accommodate the insertion string which is then copied in. If the insertion will overflow the active frame then one of two actions will take place:

   (i) if the insertion string can be accommodated within a small number of adjacent frames then rippling is implemented to distribute the text segment with insertion over the available frames. This obviates the need for a new text block to be created and the associated disk I/O.

   (ii) if the insertion cannot be accommodated as above then a new text block will have to be created and reference to it (as in its block number) and information about it (as in the number of occupied characters) should be inserted into the index block. The overflow from the active frame is then copied into the new text block.

November 15, 1983
(b) Deletion of text.
If the deletion is wholly contained within the active frame then rippling characters down within the frame will delete the desired string. (A check is made to see if the remaining characters in the text block can be distributed over the adjacent frames. If this is possible then distribution takes place and the text block is removed from the index block. This maintains text density in the text blocks, thereby reducing disk I/O for later accessing.)

If the deletion covers more than one text block then the included whole blocks may be removed from the index block and the two endpoint partial blocks will be coalesced if possible.

(c) Read the character at a particular position in the text.
This allows the text to be treated as a virtual array for all read operations. This operation returns the character which is at the specified position in the virtual array. The text block containing the desired character will be read into a frame if it is not already in one. This is necessary to access the desired character.

All processing of the text takes place at the active position, or between the active position and the position indicated by mark. This processing is achieved via the application of a set of primitive operations. (i.e. INSERT, DELETE, MOVEACTIVE, MARK, GETMARK, GETACTIVE & SEARCH.)

November 15, 1983
3.2 Implementation of Primitive Operations

INSERT(dd1,dd2) - The "Insert" primitive operator receives two dd pointers. The first, dd1, indicates the source of the insertion - a string terminated by the NULL character, or a file terminated by EOF. The second, dd2, indicates the text into which the insertion is to take place. After insertion, the active position is moved to the end of the inserted text.

StringInsert (s,dd1) -
The StringInsert function receives a pointer s to the string to be inserted and a dd pointer dd1, indicating the text into which the insertion is to take place. The text is then rippled up, starting at its active position with enough characters to accommodate the string. This rippling is done by a function CopyStr (src,dest,n). The actual insertion is performed by the function Strinsert (active,s,text).

Strinsert (p,s,t) -
The Strinsert function copies the string s' into the text t' starting at position p'. The string s' is a standard C string and is terminated by a null character.

CopyStr (src,dest,n) -
The CopyStr function copies n' bytes of storage from src' to dest'.

FileInsert (f,ddl) -
The FileInsert function receives a pointer f' to the name of the file to be inserted and a dd pointer ddl', which describes the text. If the file is successfully opened then the location of the file is determined using the system call fstat (fd,&sbuf'). The text is then rippled up, starting at its active position to accommodate the file, using the CopyStr function. Next, the file is read directly into the text buffer.

DELETE(ddl) - The "Delete" primitive operator receives a pointer to a dd - ddl', which describes the text in which deletion is to occur. The delete range is between the active position and the position indicated by mark. The characters in that range are deleted via the CopyStr function, which simply ripples down characters from the range end, over the text to be deleted and to the beginning of the range.

November 15, 1983
MOVEACTIVE(dd1,dd2,dd3) - The "MoveActive" primitive operator changes the active position of the text described by the pointer to a dd - dd3. The dd's are used as follows: dd1 describes an integer used as a switch; dd2 describes an integer, which is either added/subtracted to the current active, depending on whether the switch is set or not.

MARK(dd1) - The "Mark" primitive operator sets the mark of the text indicated by dd1 to the active position of the text indicated by dd1.

GETMARK(dd1,dd2) - The "GetMark" primitive returns the current mark value of the current text. This is done by assigning dd1's ival (see defn. of dd) the value of the text's mark index.

GETACTIVE(dd1,dd2) - The "GetActive" primitive returns the active position of the current text. This is done by assigning dd1's ival (see defn. of dd) the value of the text's active position.

GETEND(dd1,dd2) - The "GetEnd" primitive returns the current rear value of the current text. This is done by assigning dd1's ival (see defn. of dd) the value of the text's rear index.

SUBSTR(dd1,dd2,dd3,dd4) - The "Substr" primitive receives 4 pointers to dd's - dd1, dd2, dd3, dd4. The first dd is the start position of the desired substring contained in the source string described by the third dd. The second dd is the length of the desired substring and the fourth dd is the target string of the operation.

SEARCH(dd1,dd2,dd3,dd4) - The "Search" primitive receives 4 pointers to dd's - dd1, dd2, dd3, dd4. If the integer value described by dd1 is +1 then a forward search is implied and "FwdSearch (dd2,dd3,dd4)" is called. However, if the integer value is -1 then a backward search is implied and "BackSearch (dd2,dd3,dd4)" is called.

FwdSearch(dd1,dd2,dd3) -

The FwdSearch function searches from the start of the range set by the mark and active positions, of the text described by dd2, for the first occurrence of the string described by dd1. If the string is found in the range then the ival of dd3 is set to the found position, if not found the ival is set to -1.

BackSearch(dd1,dd2,dd3) -

The BackSearch function searches from the end of the range set by the mark and active positions of the text described by dd2, backwards, for the first occurrence of the string described by dd1. If the string is found then the ival of dd3 is set to the found position, if not found the ival is set to -1.

November 15, 1983
4. SCREEN CONTROL

4.1 Introduction

* Screen Control is concerned with displaying the contents of a text variable.

* The text printed on the screen is always that surrounding the active position.

* The active position is represented on the screen by the cursor, which sits on top of the first character of the second distinguished string. (Please refer to the group's specification and its definition of the abstract data type TEXT to make this perfectly clear and unambiguous.)

* All ASCII characters, including "non-printables" may be represented on the screen.

* A "message" function which displays the contents of a string allows a convenient interface between ITPL and the user.

General.

[1] Although the text is just a stream of ASCII characters, the newline and tab characters have implicit screen formatting functions. Thus, text will be displayed on the screen as a series of lines depending on the position of newline characters in the text.

[2] A line can be of any reasonable length and the screen is moved along the displayed area until the active position comes into view.

[3] The displayed area is considered to be made up of all the lines which are between the first line displayed on the screen and the last line displayed on the screen. It includes any portions of lines that extend past the left and right boundaries of the screen.

[4] The primitives involved in screen control are:

   Display[text]
   Redisplay[text]
   Message[line, col, string]

   Display is used for displaying text, and only reprints the portions of the screen that have been changed.

November 15, 1983
Redisplay reprints the entire screen, and is typically used after some form of screen corruption has occurred.

Message is described below (method [4])

4.2 Implementation

In order to minimise the time spent waiting on screen output, a Change Table has been set up to keep track of the parts of the screen that have changed so that only these changes are reprinted. Also a record of the extent of all lines in the displayed area is kept.

For every line of the displayed area the following information is kept.

(a) The position of the first character in the line
(b) The position of the last character in the line
(c) A flag representing the scope of the changes made to that line
(d) The position of the first changed character in the line

All positions are relative to the start of the displayed area.

The change flag can be:

(a) NO_CHANGE - no updating required
(b) ONE_LINE - just print one line
(c) BULK_LINES - print the current line and the remainder of the screen.

Method.

After the updating has been done the cursor is positioned to show the active position. Therefore, the four sections involved in screen control are:

[1] Updating the Change Table

[2] Updating the screen according to the Change Table

[3] Position the cursor to reflect the active position

[4] Printing a message onto the screen

November 15, 1983
[1] On each update of the text, the following parameters are passed to Screen Control by Text Control:-

(a) A flag representing either insert or delete.
(b) The position of the first character changed.
(c) The position of the last character changed.

[2] A line is printed from a position in the buffer until a newline is found. The position in the buffer is not necessarily the first character in the line. Each line in the displayed area has an entry on the Change Table, so to update a line on the screen the line's entry is read and the line is printed accordingly. If there was more than one line changed then the remainder of the screen is reprinted as well. Characters that are normally non-printable are displayed in inverse-video format. Tabs are printed by moving to the next tab position.

[3] After the screen has been updated, the cursor is moved to the position which is to the immediate right of the active position.

[4] The message function provides for a string to be printed at a specific location on the screen, over any text that may already be displayed. This allows prompts and the like to be printed without disturbing the text buffer.

After each message is printed the change table is updated so that on the next display the message will be cleared. The cursor is left at the end of the string which has been printed. If the message co-ordinates are out of range then the string will be printed at the current cursor location.

November 15, 1983
APPENDIX A :

DEFINITIONS AND TERMINOLOGY

SYMBOL TABLE

This structure contains all the data created and used by the program. The SYMBOL TABLE is implemented as a binary tree. The nodes of the tree contain the name of the symbol and a pointer to where the associated data descriptor can be found. The binary tree has no associated balancing function but relies on the premeditated ordering of the standard library (as it contains the symbols first loaded into the symbol table) to provide a well-balanced trunk upon which a reasonably well-balanced tree may grow.

DATA DESCRIPTOR

This describes the type and actual value of the data. Implemented as a variant record that contains a flag stating the type of data and the data or a pointer to the data depending on which is applicable. The allowable types of data are:

- INTEGER - an integer
- BOOLEAN - a boolean
- STRING - a string
- TEXT - (see TEXT)
- FILE - a string

TEXT

Text is a structure that contains access information to a virtual array of characters, and also contains associated positions (rear, mark and active) which define the two or three strings which constitute text.

ASSOCIATION TABLE

An array of pointers to data descriptors which is indexed by characters. The ASSOCIATION TABLE is used as a shorthand method to execute commands, thus allowing execution of macros simply by entering single keystrokes or a short sequence of keystrokes.

November 15, 1983
STANDARD LIBRARY

The standard library contains declarations of procedures which perform basic text editing actions such as those outlined below:

**Insertion of text elements**
- insert character (right of cursor)
- insert word
- insert text

**Deletion of text elements**
- delete character (left or right)
- delete word (left or right)
- delete line (left or right)
- delete text (marked position to the active position)

**Movement within text**
- move character (left or right)
- move word (left or right)
- move line (up or down)
- move to the marked position in the text.

**Displaying the text**
- implicit to all functions and shows the text.

**Searching**
- string (forward or backward)

USER LIBRARY

File of the user's personal macros that will be loaded at the start of their edit session. The user may bind certain keys to macros if desired.

November 15, 1983
APPENDIX B:

THE GRAMMAR OF ITPL

prog : procname decln_list stmt_list ENDPROC

procname : IDENTIFIER

decln_list : /* empty */
           | decln decln_list

decln : INT
       | BOOL
       | STRING
       | TEXT
       | FLE
       | MACRO

id_list : IDENTIFIER "," id_list
         | IDENTIFIER

stmt_list : /* empty */
           | stmt stmt_list

stmt : assgn_stmt

| while_stmt
| if_stmt
| prim_stmt

assgn_stmt : IDENTIFIER "ASG" exprn

while_stmt : WHILE exprn stmt_list ENDDO

if_stmt : IF exprn THEN stmt_list

ELSE stmt_list

ENDIF

prim_stmt : primname "[" exprn_list "]"

primname : "ASSOC"

| "BIND"
| "GETCHAR"
| "DISPLAY"
| "COMPILE"
| "EXEC"
| "SYSTEM"
| "INSERT"
| "DELETE"
| "SEARCH"
| "MOVE"
| "MARK"
| "GETACTIVE"
| "GETEND"
| "GETMARK"
| "SUBSTRING"
| "WRITE"
| "MESSAGE"
| "REDISPLAY"

November 15, 1983
exprn_list : /* empty */
    | exprn , exprn_list
exprn : "(' exprn ')")
    | exprn + exprn
    | exprn - exprn
    | exprn * exprn
    | exprn / exprn
    | exprn % exprn
    | exprn & exprn
    | exprn \ exprn
    | exprn < exprn
    | exprn > exprn
    | exprn LE exprn
    | exprn GE exprn
    | exprn = exprn
    | exprn NE exprn
    | exprn ` exprn ( concatenation function )
    | `- exprn %prec UMINUS
NUMBER
QSTRING
IDENTIFIER

November 15, 1983
REFERENCES

[i] - Betts, Chorvat, Constable, Fazzolare, Green : Speciﬁcation of an Editor GROUP A

[ii] - Reinfelds : Design of a Simple Editor

[iii] - M.E.Lesk & E. Schmidt : LEX A lexical analyser generator (UPM vol 2B)

[iv] - S.C.Johnson : YACC Yet another compiler compiler. (UPM vol 2B)

November 15, 1983
NAME
qedit - an interactive text editor

SYNOPSIS
qedit [ filename ]

DESCRIPTION
On invocation qedit loads the macros found in the standard library "slib" and according to user response determines the mode of operation. There are at present four modes designed primarily to show the capabilities of qedit. These are a screen based editor, a text formatter, a small mathematical interpretation program and a Pascal formatter. Qedit is an editor with the ability to define macros. The editor, formatter, mathematics library and Pascal formatter are written as macros. This then leads to the ability to define your own macros to do your own tasks. The filename is optional and if given is the name of the file that is to be loaded into qedit's text buffering system. If not given then you are simply thrown into the default buffer, "current".

FILES
appropriate files in /cs/300/jjc/PROJECT/FAZZ/HANDIN/*

SEE ALSO

BUGS
There is no serious attempt at error recovery from an erroneous macro definition. The most likely thing that will happen is that the parser will spin into an infinite loop until it causes the QEDIT to dump core. Therefore users are advised to test macros before use.
QUEDIT - USER MANUAL

GROUP A

David Betts
Trevor Chorvat
John Constable
Michael Fazzolare
Craig Green

Wollongong University
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. QUEDIT - An Interactive Text Editor</td>
<td></td>
</tr>
<tr>
<td>1.1 What is QUEDIT</td>
<td>3</td>
</tr>
<tr>
<td>1.2 How to use QUEDIT</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 The cursor</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Inserting Text</td>
<td>3</td>
</tr>
<tr>
<td>1.2.3 Deleting Text</td>
<td>4</td>
</tr>
<tr>
<td>1.2.4 Cut/Copy and Paste</td>
<td>4</td>
</tr>
<tr>
<td>1.2.5 Searching for Strings</td>
<td>4</td>
</tr>
<tr>
<td>1.2.6 Replacing the Strings</td>
<td>5</td>
</tr>
<tr>
<td>1.2.7 Going to a specific line</td>
<td>5</td>
</tr>
<tr>
<td>1.2.8 Overtyping Text</td>
<td>5</td>
</tr>
<tr>
<td>1.2.9 Re-displaying the Screen Window</td>
<td>5</td>
</tr>
<tr>
<td>1.2.10 Executing macros</td>
<td>5</td>
</tr>
<tr>
<td>1.2.11 Switching to another Text Buffer</td>
<td>5</td>
</tr>
<tr>
<td>1.2.12 Making new macros while in QUEDIT</td>
<td>6</td>
</tr>
<tr>
<td>1.2.13 UNIX Commands</td>
<td>7</td>
</tr>
<tr>
<td>2. FAZZPAS - The Text Processing Language</td>
<td></td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>8</td>
</tr>
<tr>
<td>2.2 FAZZPAS Statements used in Procedures</td>
<td>10</td>
</tr>
<tr>
<td>2.3 FAZZPAS Expressions</td>
<td>17</td>
</tr>
<tr>
<td>2.4 FAZZPAS Functions</td>
<td>20</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td></td>
</tr>
<tr>
<td>An example of what can be done with an Interactive Text Processing Language</td>
<td>28</td>
</tr>
</tbody>
</table>
1. QUEDIT - An Interactive Text Editor

1.1 What is QUEDIT?

QUEDIT is an interactive, screen based text editor, written in the ITPL [1] FAZZPAS. The commands of QUEDIT are written as FAZZPAS macros, and hence may be executed by name, or by binding [1] a macro to a single key. This then allows the macro to be executed by a single key stroke.

1.2 How to use QUEDIT.

To begin editing a file, type the UNIX command

```
q filename
```

The first 23 lines of the file will appear on the screen. When using `q`, the screen acts like a sheet of paper, onto which you can type text with the keyboard. Every keystroke has an immediate visible effect and so the image on the screen always corresponds exactly to the state of the text.

To leave QUEDIT, type CTRL-D, and QUEDIT will prompt you as to whether you would like to save the copy of the file it has been working on, as the original. You should reply `y` or `no`, anything else returns you to the editor.

1.2.1. The cursor

All changes to the text occur at the cursor position, which is always between two characters.

Cursor movement is achieved via the three keys labelled with arrows, which move the cursor one position in the indicated direction (LEFT,RIGHT,UP), and the LINEFEED key, which moves the cursor DOWN by one position. Any of these keystrokes preceded by the ESCAPE key will move the cursor 10 spaces in the indicated direction.

Any attempt to move the cursor `off` the screen will cause the screen window to move in the indicated direction, if there is text there. Lines can be of any length.

1.2.2. Inserting text

When you type a visible character it appears on the screen at the cursor position, and the text to the right of that position is moved along one position.

Control characters may be inserted by executing the macro `\texttt{inspecial}` (see later), which takes the next character entered and inserts it into the text as above. All control characters in the text are shown in inverse video.

The RETURN key inserts a newline character at the cursor position, the cursor then moves down to the beginning of the next line.
A file may also be inserted into text at the cursor position, by going to the command line (type CTRL-@), entering the name of the file, and then typing CTRL-A. To rectify an error made in typing the string or file name you can use the <CLEAR> key.

1.2.3. Deleting text

The RUBOUT key deletes the character to the right of the cursor.

The CLEAR key deletes the character to the left of the cursor.

CTRL-L will delete from the cursor to the end of the line.

CTRL-W will delete from the cursor to the end of the word to the right of the cursor.

CTRL-T will delete from the cursor to the start of the word to the left of the cursor.

Whenever text is deleted using one of CTRL-L, CTRL-W, CTRL-T, the deleted text is placed into a delete buffer, called 'delbuf', which may be inserted into the text by typing CTRL-N.

1.2.4. Cut/Copy and Paste

The HOME key marks the text at the current cursor position, and a message is displayed saying so. If you move the cursor somewhere else, and then type CTRL-K or CTRL-C, the marked region of text, between the mark and the current cursor position, is placed into a paste buffer called 'paste'. This buffer may then be inserted into the text by typing CTRL-P.

Note, CTRL-K also deletes the marked region after placing it into the paste buffer, while CTRL-C merely places the marked region of text into the buffer without deletion.

1.2.5. Searching for Strings

To search for a string, go to the command line, enter the string, and then type CTRL-F to search forward or CTRL-B to search back. If the string is found, then the cursor is positioned at the start of the string in the text, otherwise a message is displayed and the terminal "beeps". The find string is saved, so by just typing CTRL-F or CTRL-B, without going to the command line, will search for what is already in the find string.

1.2.6. Replacing strings

When the cursor is positioned at the start of the find string in text, go to the command line, enter the replace string, and type CTRL-R for a single replace or CTRL-^ for a global replace from the current cursor position.
1.2.7. Going to a specific line

Go to the command line, type in the line number, and then type CTRL-G. If the line exists, then the cursor is positioned at the start of that line. Otherwise, you are positioned at the start of the last line of the text.

1.2.8. Saving text in a file

You can save the current state of the text without leaving the editor by typing CTRL-S, which will save the current buffer of text in the last entered file name. The default file name is that with which you started the editing session. To save the text in another file, go to the command line, enter the file name, then type CTRL-S.

1.2.9. Overtyping text

To switch QUEDIT to overtype mode, type CTRL-O. To switch back to insert mode, type CTRL-O again.

1.2.10. Re-displaying the screen window

If the screen gets overwritten by messages from other processes, simply type CTRL-V to re-display the screen window.

1.2.11. Executing macros

To execute a FAZZPAS macro, go to the command line, type the name of the macro, then type CTRL-E.

1.2.12. Switching to another Text Buffer

When QUEDIT is initially invoked, the user edits a buffer called 'main', into which his/her file is inserted. QUEDIT allows the user to switch to other buffers, creating them if necessary, for editing.

To switch to another buffer, go to the command line, enter the buffer name, then type CTRL-Q. If the buffer exists, then its text is displayed on the screen, and you may now edit the buffer. If the buffer does not exist, then you are prompted to see if you would like the buffer created. If so, the buffer is created, and you have a new buffer to edit.

This feature is particularly useful for editing other files: you simply switch to a new buffer, and then insert the file.

The standard buffers used by q are: 'main', 'paste' and 'delbuf'. Hence, it is possible to edit the paste and delete buffers, before inserting them back into text.

1.2.13. Making new macros while in QUEDIT

QUEDIT supplies a standard set of editing commands/macros. It also allows you the flexibility of changing the value of its constants (e.g. 'movsize'), re-
binding the keys to other macros (eg CTRL-Q to insert a CTRL-Q character), and to make up new macros/commands, store them, and execute them.

To change the value of a q constant (eg 'movelsize'), type CTRL-_ (control underscore), and then enter:

\[ \text{movelsize := 20} \]

then press RETURN. This will execute a macro to reset the value of 'movelsize'.

To re-bind a key, say CTRL-Q to insert a CTRL-Q character, type CTRL-_ , then enter:

\[ \text{bind["#Q",inschar]} \]

then press RETURN.

To make up a new macro

i) switch to a new buffer, say mybuf,

ii) edit mybuf so that it contains the statements of the macro,

iii) type CTRL_-_ then enter:

\[ \text{macro mymacro;compile[mybuf,mymacro]} \]

then press RETURN.

Now the name of your macro is known to QUEDIT, and so it may be executed by name, or bound to a key.

1.2.14. UNIX commands

To execute UNIX commands from within QUEDIT, execute the macro 'callsystem', which places you in the UNIX shell.
2. FAZZPAS - The Text Processing Language.

2.1. Introduction to FAZZPAS.

Like any language, FAZZPAS has its own vocabulary and syntax - words and the rules for putting them together. However, because FAZZPAS is a text processing language, it has a few features not normally found in a programming language, e.g. the datatype TEXT.

FAZZPAS Statements.

A FAZZPAS statement is a string of FAZZPAS keywords, FAZZPAS names, and special characters and operators. Only type definitions end in a semi-colon.

Here are some examples of FAZZPAS statements:

\[ \text{while ( a < b ) & ( c \geq d )} \]
\[ \text{procedure fred} \]
\[ \text{integer i;} \]

FAZZPAS keywords

A FAZZPAS statement begins with a keyword that identifies the kind of statement it is. The FAZZPAS statements shown above do not form a FAZZPAS program but are single FAZZPAS statements referred to by their keywords as a WHILE statement, a PROCEDURE statement and an INTEGER declaration. 

(Declaration types can be any of the following: INTEGER, BOOLEAN, TEXT, FILE, MACRO and STRING.)

FAZZPAS Names.

The first character of a FAZZPAS name must be a letter (A-Z, a-z) and can be followed by an arbitrary number of letters, numbers (0-9) or the underscore character (_).

Blanks may not appear in FAZZPAS names, and special characters (for example, $,#) except for the underscore, are not allowed.

Special Characters and Operators.

Only declaration FAZZPAS statements end with a semi-colon. Other special characters and operators illustrated in the statements above are the right and left parentheses "(",")"; the left and right brackets "[","]"; addition operator or plus sign "+", less than "<", and greater than or equal "ge" and the and operator "&".

WRITING FAZZPAS PROGRAMS.

The FAZZPAS language allows flexibility in the way
FAZZPAS statements are written in a program. A program is made up of a procedure name, the declarations, the statements and then an end of program statement. FAZZPAS programs can be saved away in a user library file which is available for use in a later editing session.

Required Spacing

FAZZPAS statements may begin in any column of a line and several statements may appear on the same line. You need at least one blank between each separate item in a FAZZPAS statement as in the statements above. Some special characters, such as the colon equal sign after a word, can take the place of a blank, although blanks are always allowed.

For example, in the statement

\[ \text{count:=count+1} \]

you do not need a blank before or after the colon equals sign or before or after the plus sign, because the colon equals and the plus sign are special characters, but you can also have any number of extra blanks. The statement

\[ \text{count := count + 1} \]

is equivalent to the statement above.

Comments.

Comments of the form \{ This is a comment \} may appear within FAZZPAS statements wherever a single blank may appear. For example, the statement

\[ \text{procedure delword \{ delete the next word \}} \]

is a valid FAZZPAS statement.

Strings.

Strings are made up of any legal character (i.e. any printable character) between single quotes. If control characters are needed in a string then the following escape sequences are allowed: #[A-Z] which are mapped into their control characters. To get a # in a string use ## and to get a ' in a string use #'.


2.2. FAZZPAS Statements used in Procedures.

The FAZZPAS statements and declarations that are used in PROCEDURES are described alphabetically in this chapter.

Assignment Statement

BOOLEAN Declaration

ENDIF Statement

ENDPROC Statement

ENDWHILE Statement

FILE Declaration

IF Statement

INTEGER Declaration

MACRO Declaration

PROCEDURE Statement

STRING Declaration

TEXT Declaration

WHILE Statement
Assignment Statement.

Assignment statements evaluate an expression and store the result in a variable. Assignment statements have the form:

```plaintext
variable := expression
```

The terms in the assignment have these definitions:

- **variable**: declared at the top of the program
- **expression**: is one or more variable names and constants linked by operators and by parentheses where appropriate.

For example, the statement:

```plaintext
a := b + c
```

adds the previously defined integer values of `b` and `c` and puts the result into `a`; and,

```plaintext
a := 1
```

assigns the value 1 to the previously defined integer variable called `a`.

BOOLEAN Declaration.

BOOLEAN declarations declare a variable of type boolean. BOOLEAN declarations have the form:

```plaintext
boolean continue;
```

which defines the variable `continue` of type boolean.

ENDIF Statement.

The ENDIF statement is the last of the FAZZPAS statements that make up an IF THEN/ELSE group. The form of the ENDIF statement is:

```plaintext
endif
```

An ENDIF statement must end every IF THEN/ELSE group in your FAZZPAS program. Nesting of IF THEN/ELSE's is allowed.

For example an IF THEN/ELSE group is shown below:
if
    pos := pos2
else
    pos := pos3
endif
endif

and an example without an ELSE clause is:
    if pos1 < 0 then pos1 := 0 endif

ENDPROC Statement

The ENDPROC statement is the last of the FAZZPAS statements that makeup a PROCEDURE group. The form of the ENDPROC statement is:

    endproc

An ENDPROC statement must end every PROCEDURE group in your FAZZPAS program. For example a simple PROCEDURE group is shown below:

    procedure fred
        :
        FAZZPAS statements
        :
    endproc

ENDWHILE Statement.

The ENDWHILE statement is the last of the FAZZPAS statements that make up a WHILE group. The form of the ENDWHILE statement is:

    endwhile

An ENDWHILE statement must end every WHILE group in your FAZZPAS program. For example a simple WHILE group is shown below:
while ( pos = act )
    :
    FAZZPAS statements
    :
endwhile

IF statement.

In FAZZPAS there is a conditional IF statement, written with a THEN clause, which is used to conditionally execute a FAZZPAS statement. An optional ELSE statement gives an alternative action if the THEN clause is not executed.

IF - THEN and IF - THEN/ELSE Statements.

FAZZPAS evaluates the expression following the IF. When the expression is true, the statement following THEN is executed. When the expression is false, FAZZPAS ignores the statement following THEN and executes the ELSE statement immediately following the IF. If no else statement is present, FAZZPAS executes the next program statement.

The form of the IF - THEN statement is:

IF expression THEN statement ENDIF

where

expression is any valid FAZZPAS expression. See Chap. 2.  "FAZZPAS Expressions", for the way FAZZPAS evaluates expressions.

statement may be an assignment statement, an IF statement, a WHILE statement or a FAZZPAS function.

For example, the statement

if posl < 0 then posl := 0 endif

assigns the value 0 to an integer variable named posl when posl is less than zero.

The ELSE statement can immediately follow an IF - THEN statement to specify a statement that is to be executed when the condition of the IF is false. The form of the ELSE statement is:

ELSE statement
where

statement is either an assignment statement, an IF
statement, a WHILE statement or a FAZAPAS
function.

These statements:

```
if pos1 < 0 then
  pos1 := 0
else
  pos1 := 1
endif
```

assign 1 to the integer variable named pos1 when pos1 is not
less than zero.

INTEGER Declaration

INTEGER declarations declare a variable of type
integer. INTEGER declarations have the form:

```
integer count;
```

which defines the variable 'count' of type integer.

MACRO Declaration.

MACRO declarations contain macros. MACRO statements
have the form:

```
macro XX;
```

which defines 'XX' as a macro.

STRING Declaration.

STRING declarations declare variables of type string.
STRING statements have the form:

```
string alphal;
```

which defines 'alphal' as a string.
TEXT Declaration

TEXT declarations declare a variable of type text. TEXT declarations have the form:

text current;

which defines the variable "current" of type text.

WHILE Statement

The WHILE statement specifies that the statements following the WHILE are to be executed as a unit until a matching ENDWHILE statement appears. The statements between the WHILE and the ENDWHILE statement are called a WHILE group. Any number of WHILE groups can be nested.

You can execute the statements in a WHILE group repeatedly while a condition holds using the WHILE statement. The form of the WHILE statement is:

WHILE expression statement ENDWHILE

where

expression is any expression. (See Chap. 3, "FAZZPAS Expressions", for the way FAZZPAS evaluates expressions.)

statement may be an assignment statement, an IF statement, a WHILE statement or a FAZZPAS function.

The expression is evaluated at the top of the loop before the statements in the WHILE loop are executed. If the expression is true, the WHILE group is executed.

These statements repeat the loop as long as N is less than 5.

N := 0
while ( N < 5 )
    N := N + 1
endwhile
2.3. FAZZPAS Expressions.

Introduction.

An expression is a sequence of operators and operands forming a set of instructions that are performed to produce a result value. The operands are variable names. The operators are special character operators and grouping parentheses. An expression can reduce to a string, an identifier, a number or a boolean value (i.e. true or false).

Expressions can be simple (using only one operator) or compound (using more than one operator). The following are examples of expressions:

\[
\begin{align*}
N + 1 \\
3 \\
N > 10 \\
( pos > 4 ) & ( pos < 20 )
\end{align*}
\]

Expressions can be used in PROCEDURES for conditional processing, calculating new values and assigning new values.

FAZZPAS OPERATORS.

FAZZPAS operators are symbols that request a comparison, logical operation or arithmetic calculation.

Arithmetic Operators.

Arithmetic operators indicate that an arithmetic calculation is to be performed. The arithmetic operators are:

* multiplication

/ division

+ addition

- subtraction

% modulus

Comparison Operators

Comparison operators propose a relationship between two quantities and ask FAZZPAS to determine whether or not that relationship holds. If it does hold (in other words, if it is true), the result of carrying out the operation is the value 1; if it does not hold (in other words, if it is false), the result is the value 0. The comparison operators are:
Consider the expression \( A \leq B \). If \( A \) has the value 4 and \( B \) has the value 3, then \( A \leq B \) has the value 0 (false). If \( A \) is 5 and \( B \) is 9, then the expression has the value 1 (true). If \( A \) and \( B \) each have the value 47, then again the relationship holds and the expression assumes the value 1.

Comparison operators appear frequently in IF statements. For example:

```pascal
if X < Y then
  C := 5
else
  C := 12
endif
```

Strings can also be compared. They are compared alphabetically. For example:

```pascal
name1 := 'happy'
name2 := 'sad'
if name1 < name2 then
  ::
  FAZZPAS statements
  ::
endif
```

Logical Operators.

Logical Operators, also called Boolean operators, are usually used in expressions to link sequences of comparisons. The logical operators are:
& AND

or OR

If both of the quantities surrounding an AND are 1 (true), then the result of the AND operation produces a 1; otherwise the result is 0. For example, the expression:

A < B & C > 0

is true (has the value 1) only when both A < B is 1 (true) and C > 0 is 1 (true) that is, when A is less than B and C is positive.

If either of the qualities surrounding an OR is 1 (true) then the result of the OR operation is 1 (true); otherwise the OR operation produces a 0. For example, the expression

A < B or C > 0

is true (has the value 1) when A < B is 1 (true) regardless of the value of C. It is also true when the value of C > 0 is 1 (true), regardless of the values of A and B. It is true, then, when either or both of these relationships hold.

Other Operators.

The operator in this category is concat, which concatenates two string variables. For example, if the string variable COLOUR has a value of BLACK, and the string variable NAME has a value of JACK, then:

GAME := COLOUR concat NAME

results in GAME having a value of BLACKJACK.
2.4. FANZPAS FUNCTIONS

Introduction

A function is a routine that returns a value computed from arguments. Each function has a keyword name. To invoke a function, write its name and then the argument or arguments for which the function is to be performed enclosed in square brackets:

FUNCTIONNAME[argument/s]

Some functions only require 1 argument whereas others require up to four arguments. The supplied functions are:

ASSOC  MARK
BIND  MESSAGE
DELETE  MOVE
DISPLAY  REDISPLAY
GETACTIVE  SEARCH
GTCHAR  SUBSTR
GETEND  SYSTEM
GETMARK  TELL
INSERT
FUNCTION DESCRIPTIONS.

BIND Function.

The BIND function takes two arguments, the first being the string to be bound and the second being the macro that it is to be bound to. The form of the BIND function is:

BIND[string1,macroname]

- string1 can be any ASCII character [ A,...,Z, a,...,z, 0,...,9 ], special characters ( #,$ etc. ) or control characters.

- macroname is the name of the FAZZPAS procedure that is to be executed when a specific key is pressed.

For example, the statement

bind["#B",insbegin]

binds <ctrl B> to a macro called "insbegin". ( i.e. "insbegin" is executed when <ctrl B> is pressed on the keyboard.

COMPILE Function.

The COMPILE function takes macro text and produces the appropriate executable code. The form of the COMPILE function is:

COMPILE[filename,macroname]

DELETE Function.

The DELETE function deletes a string from the marked position to the active position. The DELETE function only receives one argument that being the text in which the deletion is to occur. The form of the DELETE function is:

DELETE[buf1]

where buf1 is defined as type text.

DISPLAY Function.
The DISPLAY function takes one argument that being the text buffer that you want displayed and displays any changes made to that buffer. The form of the DISPLAY function is:

\[ \text{DISPLAY}[\text{current}] \]

EXEC Function.

The EXEC function executes a compiled piece of FAZZPAS code. It takes one argument which is the macro to be executed. The form of an EXEC statement is:

\[ \text{EXEC}[\text{macroname}] \]

GETACTIVE Function.

The GETACTIVE function finds the current active position of the given buffer name. The GETACTIVE function takes two arguments, the first a positional parameter in which an integer value of the current active position is placed and the second is the name of the text buffer. The form of the GETACTIVE function is:

\[ \text{GETACTIVE}[\text{pos,buffer1}] \]

- pos is defined as type integer, and
- buffer1 is defined as type text.

GTCHAR Function.

The GTCHAR function gets a character from standard input. The form of the GTCHAR function is:

\[ \text{GTCHAR}[\text{targetstring}] \]

which puts the character into the string specified.

GETEND Function.

The GETEND function finds the last character position
of the given buffer name. The GETEND function takes two arguments, the first a positional parameter in which an integer value of the last character position is placed and the second is the name of the text buffer. The form of the GETEND function is:

\[ \text{GETEND}\{\text{pos,bufferl}\} \]

- \text{pos} is defined as type integer, and
- \text{bufferl} is defined as type text.

**INSERT Function**

The INSERT function inserts marked text into another text buffer. An INSERT function takes two arguments which are expressions that represent the string to be inserted and an expression representing the text, the string that is to be inserted into. The form of the INSERT function is:

\[ \text{INSERT}\{\text{paste, current}\} \]

- \text{paste} is the marked piece of text and is declared as type text, and
- \text{current} is the text buffer to be inserted into and is also defined as type text.

For example, the statement

\[ \text{insert}\{\text{paste, current}\} \]

inserts the string of text called 'paste' into another string of text called 'current'.

**MARK Function.**

The MARK function sets the mark of the text indicated by the argument to the active position. The form of the MARK function is:

\[ \text{MARK}\{\text{bufl}\} \]

- \text{bufl} is the name of the string of text in which you want to mark the current active position (i.e. where the cursor is). For example, the statement

\[ \text{mark}\{\text{current}\} \]
marks the current active position of the string of text named "current". "Current" is declared as type text.

MESSAGE Function.

The MESSAGE function prints a message. It takes 3 arguments, the first is the line number on the screen that you want the message printed, the second is the column on the screen that you want to start printing at and the third is the message you want printed in the form of a string. The form of the MESSAGE statement is:

MESSAGE[23,2,"This is a message"]

which prints "This is a message" on line 23 starting at column 2.

MOVE Function.

The MOVE function changes the active position in the text described by the third argument. The first argument describes an integer which is used as a switch, the second argument describes an integer which is either added/subtracted to the current active position depending on whether the switch is set or not. The form of the MOVE statement is:

MOVE[vall,val2,bufl]

- vall is either 1 or 0. If vall = 1 then the value of val2 is added to the current active position and the cursor moves to that position or if vall = 0 then the value of val2 is the new active position in the text.

- val2 is an integer value used to find the required active position.

- bufl is text in which you want to move the active position.

For example, the statement

move[0,pos,current]

moves the active position of the string of text named "current" to the position in the text denoted by pos, whereas the statement

move[1,-1,current]
moves the active position in the string of text back one position.

REDISPLAY Function.

The REDISPLAY function reprints the whole screen. The form of the REDISPLAY function is:

\[
\text{REDISPLAY}[\text{current}]
\]

which displays "current" of type text.

SEARCH Function

The SEARCH function receives four arguments. If the integer value described by the first argument is a 1 then a forward search is implied whereas a -1 designates a backward search. The second argument describes the string being searched for, the third argument describes the text to be searched and if the string is found then its position in the text is placed in argument 4 otherwise argument 4 is set to -1. The form of the SEARCH function is:

\[
\text{SEARCH}[\text{arg1}, \text{arg2}, \text{arg3}, \text{arg4}]
\]

- \text{arg1} = 1 - denoting a forward search, or
- \text{arg1} = -1, denoting a backward search
- \text{arg2} string being searched for
- \text{arg3} text in which string is being searched for
- \text{arg4} is returned the position in the text of the first occurrence of the string or -1 if the string is not found.

For example, the statement

\[
\text{search}[1, \text{"six"}, \text{current}, \text{pos}]
\]

searches forward from the active position, for the string "six" in the text buffer named "current". Pos will contain the position in the text of the first occurrence of the string "six" or -1 if the string "six" is not found. Another example is the statement

\[
\text{search}[-1, \text{"N"}, \text{current}, \text{pos}]
\]

searches backward for the first occurrence of a newline character in the text named "current". The position in the text of the newline character found is placed in pos or -1
SUBSTR Function.

The SUBSTR function receives four arguments. The form of the SUBSTR function is:

```
SUBSTR[arg1, arg2, arg3, arg4]
```

- `arg1` startpos in string. (0 is first position)
- `arg2` number of chars required
- `arg3` source string
- `arg4` target string

For example the statement

```
substr[2,3,str1,str2]
```

operating on `str1 = "abcdef"` would return `"cde"` in `str2`.

SYSTEM Function.

The SYSTEM function returns you to the UNIX shell. The form of the statement is:

```
SYSTEM[]
```
APPENDIX A:

PASCAL FORMATTING AID.

The PASCAL FORMATTING AID (PFA) is a modified version of QUEDIT. It exemplifies what can be done with an ITPL.

The main difference between QUEDIT and the PFA is in the binding of the keys. The PFA contains all of the features of QUEDIT, plus a few additional macros, as described below.

1. Starting a new procedure

Type CTRL-P, and you will be prompted for the procedure name. Enter the name, and press RETURN. The PFA will then insert the procedure heading into the text, at the current indentation level.

2. Declaring constants

Type CTRL-C, and you will be prompted for the constant name. Enter it, and press RETURN. You will then be prompted for the constant value. Enter it, press RETURN, and will be prompted for a comment. If no comment is desired, then simply press RETURN, else enter the comment and press RETURN.

3. Declaring types

Type CTRL-T, and you will be prompted for the type name. Enter it, and press RETURN. You will then be prompted for the type value. Enter it, press RETURN, and will be prompted for a comment. If no comment is desired, then simply press RETURN, else enter the comment and press RETURN.

4. Declaring variables

Type CTRL-V, and you will be prompted for the variable name. Enter it, and press RETURN. You will then be prompted for the variable type. Enter it, press RETURN, and will be prompted for a comment. If no comment is desired, then simply press RETURN, else enter the comment and press RETURN.

5. Starting a new block

Type CTRL-B, and you will be prompted for a comment. If no comment is desired, press RETURN, else enter the comment and press RETURN. The PFA will indent the text by the tabsize, and insert

\begin{comment}

into the text.

6. Terminating a block
Type CTRL-E, and
   end

will be inserted into the text at the current indentation level, and the indentation level will be decreased by one.

7. Entering a newline

Pressing RETURN will take you to the next line and place you at the current indentation level.

8. Inserting a comment

Type \{ , and you will be prompted for the comment. If no comment is desired, press RETURN, else enter the comment and press RETURN.

The user should note that the PFA is a formatting aid.

NO ERROR CHECKING IS PERFORMED