in

COLLABORATORS

	TITLE : in		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 6, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

in

1.1 JEd V2.07

```
                    JEd V2.07 16-Jan-93
         Yet another programmer's editor
         Copyright (c) 1992-3 John Harper
```

Contents:

```
                Contact Address
```

1.2 Introduction

```
Introduction.
*************

JEd is a text editor best suited to programming, it has no text formatting
capabilities (except for a dumb wordwrap). I wrote it because I found that
no available editor suited me perfectly - this one does (maybe). You may
have seen my previous attempt at this goal, JEd 1.something, version 2 is
similar in some respects but completely different in others.

If you are looking for a straightforward, user-friendly editor -- look
somewhere else :) but, if you want a non-restrictive editor which can be
made to do almost anything you want then read on...

a quick feature list:
    * totally customizable, all keys may be made to do anything,
    user-definable menu bar, etc...
    * powerful programming language
    * multi-file/multi-view editing
    * number of windows is only limited by memory
    * clipboard support (cut/paste on any unit)
    * any window can have any (non-proportional) font
    * maximum number of lines in a file is 2147483648, each line can have up
    to 32768 characters in it.
    * fast enough, even when working with large files on a 68000 cpu
    * line-undo feature
    * windows can open on any public screen (usually the Workbench)
    * full Un*x-style regular expression support (searches & substitutions)

 JEd needs system 2.0 or later.
```

1.3 Disclaimer

```
Disclaimer.
***********

THIS PROGRAM IS PROVIDED ON AN ‘AS IS’ BASIS, NO WARRANTIES ARE MADE, EITHER
EXPRESSED OR IMPLIED. IN NO EVENT WILL I, JOHN HARPER, BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING FROM ANY USE
OR MISUSE OF THESE PROGRAMS. THE ENTIRE RISK AS TO THE RESULTS AND
PERFORMANCE OF THIS PROGRAM IS ASSUMED BY YOU.
```

1.4 Distribution

```
Distribution.
*************

Distribute these files as much as you want, from now on and until further
notice they are classed as freeware and may not be sold for more than a
nominal fee to cover disks, etc.

You should be able to get the latest version by anonymous ftp from
amiga.physik.unizh.ch or any of its mirrors, probably in the aminet
directory util/edit.

I certainly won't refuse any donations sent to me but there is no
obligation.

If you want the latest version send me a disk and return postage and you'll
get it. If you just send a disk you probably won't see it again (I'm only a
poor student you know :-)
```

1.5 Installation

```
Installation.
*************

Copy the executables, jed, makerefs and pubman, to somewhere in your path.
Create a directory s:jed and copy the contents of the macros directory into
it.

These system libraries are needed in libs:
    asl.library
    diskfont.library
    iffparse.library

The clipboard.device is usually needed in devs:
If you want to use the ARexx interface ARexx should be running.

Note:
    If you want to increase the editor's scrolling speed make sure that the
    commodities.library hasn't installed it's input handler. As long as no
    program has the library open it should be OK, ie, don't have any
    commodities installed, use DMouse or something similar instead. If you
    don't believe me try scrolling through a file with no cx, then 'run
    exchange' and try it again. I think that you only need to do this if you
    have a 68000 cpu, it certainly works on my Amiga.
```

1.6 Startup

```
                Startup.
********

JEd can be run from the CLI or the Workbench, no files or options can be
specified when running from Workbench but CLI command line has this format,
    FILES/M,PUBSCREEN/K,DTAB/N/K,CD/K
```

the individual keywords represent,

```
    FILES files to load/create
    PUBSCREEN public screen to open on
    DTAB  size of tabs in files
    CD    editor's current directory (see
                cd
                )
```

The first window's preferences are loaded from the file s:jed.config, this file is created whenever the last editor window is closed. As well as containing all options set by the 'setpref' command it stores the dimensions of the last window as well.

After loading any files specified by the command line the script file jed-startup is executed, this can contain any normal JEd command strings, it is looked for first in the current directory and then (only if not already found) in the s:jed/ directory and then the s: directory. This file is normally used to create keybindings, macros, maybe load a menu if you want, etc...

If you don't supply a startup file the editor will be totally unusable (but you can quit :-) since initially all keys (except the cursor keys) just insert the characters that they are mapped to by the system keymap.

1.7 Command Language

```
                Command Language.
*****************
```

Although it is possible to use JEd without understanding its script language *much* more power can be got with a full understanding. That is what this chapter tries to give.

There are only two data types to deal with, these are
string
s and
number
s.
The language heavily enforces data typing in that it is (almost) impossible to pass a number when a string is needed, if you do manage to do this (maybe when
format
ting a string) you could well pay a visit to the guru's replacement.

Variables, on the other hand, have less typing, they assume the type of whatever is assigned to them.

Programs are stored in
command string
s, which are built up from

clause
s.

See also,

Value
,
String
,
Number
,
Command String
,
Clause
.

1.8 Value

Value.
======

A "value" is an item of data, either a
number
 or a
string
.

See also,

Number
,
String

1.9 String

String.
=======

A "string" is an array of characters.

1.10 Number

Number.
=======

A "number" is a signed 32-bit integer, ie, any whole number from -2147483648
through zero to +2147483647.

1.11 Command String

```
                Command String.
===============

A command string is a collection of
                clause
                s, one after the other. The
                value
                of the command string is the value of the last clause in the  ↩
                   string.

Some examples of valid command strings are,

    (
                settitle
                 `foo')
    (
                global
                 `date' (
                info
                 `date'))(settitle (date))

Note that
                comments
                 may be inserted between any clauses in a command string.

Also note that just because I have called command strings _command_ strings
they don't have to contain symbol (or command) clauses. It is totally
acceptable to have a string of string clauses or any other clause type. This
is often very useful, for example by putting
                string
                 or
                number clause
                s into
the command strings of the
                if
                 command you can simulate the C languages
ternary operator, ie,

(
                settitle
                 (if (
                getpref
                 `scrollhack') ``s'hack is on'' ``s'hack is off''))

this sets the
                title bar
                 to a string representing the status of the
'scrollhack' preference option.

See also,

                Clause
                ,
                Value
```

,
Comments

1.12 Comments

```
                Comments.
=========

Comments may be inserted between any clauses in a
                command string
                 (the same
is also true about whitespace characters).

Comments are introduced by the ';' character, when a ';' is encountered the
rest of the current line is disreguarded. ie,

    (move `l' 20)   ; this is a comment,
    (settitle     ; and so is this.
  `foobar')

Actually there is a problem with comments, if they are inside a string, eg,
in a command string which is to be a macro the comments will not be stripped
until the command string is executed. If the comment contains un-escaped
quotes or braces the string will be prematurely terminated and things will
go wrong. eg,

    (macro `amacro'
    {
  (settitle `hello')  ; this is ok until I put in a ' or a ` or {}
    })

to fix this you can put in escape characters like,

    (macro `amacro'
    {
  (settitle `hello')  ; this is ok until I put in a \' or a \` or \{\}
    })

alternatively just don't put these characters in comments inside strings.
```

1.13 Clause

```
                Clauses.
========

Clauses are the most elementary part of a
                command string
                , every clause has a
clearly defined
                value
                 (even if that value is defined as being void, ie, no
value).
```

There are several different types of clauses, each with their own syntax structure, these are,

Symbol Clause

String Clause

Number Clause

Character Clause

Null Clause
See also,

Command String
,
Value

1.14 Symbol Clause

```
                 Symbol Clause.
--------------

Two possible types of
                 symbol
                 s, variables and commands, so two similar clause
syntaxes,

variable clauses:

    (symbol_name)
or
    symbol_name

    eg, If you have set up a variable, foo, to access it's contents you
    would use the clause,
  (foo)
    or,
  foo

    Note that if no symbol of "symbol_name" is found internal to JEd, the
    editor will look to see if a standard DOS variable (local or ENV:) of
    that name exists. If so, the value of the clause will be whatever that
    variable contains and will have type
                 string
                 .

command clauses:

    (symbol_name optional_argument_clauses)
or,
    symbol_name
    eg, To use the command
                global
                 to create a variable called foo (as
    accessed above) you would need the clause,
  (global `foo' @)
    The '@' (
                null clause
                ) is used so that the contents of the variable are
    void.

    Note that if no symbol of name "symbol_name" is found internal to JEd,
    the REXX: directory will be checked for a file called "symbol_name.jed".
    If this file is found it is assumed to be an ARexx macro for JEd and it
    will be executed accordingly. The value of the clause will be non-zero
    if the macro was invoked successfully.

    Note that command clauses don't have to have any arguments, so they can
    be similar to variable clauses.

    The value of a symbol clause is the value of either the referenced
    variable or the value returned by the executed command.

See also,

                Clause
                ,
                Symbol
```

1.15 String Clause

```
                String Clause.
--------------

A string clause is defined syntactically as,
    `string-value'
or
    {string-value}

The value of this clause is "string-value" (of type
                string
                ).

The quotes or braces nest so if you gave a clause of
    `an example of `nested \tstrings'!'
you would get a value of,
    an example of `nested \tstrings'!

At present quotes and braces are considered equivalent so you can't surround
one by the other (This may change soon).
The braces are provided for two reasons,
    1) So that the quotes don't get screwed up by ARexx.
    2) If you use braces to enclose command strings everything is much easier
       to read in complex statements (macros, loops, etc)

IMPORTANT:
```

Unlike similar languages (LISP, the Wack script language, etc) all clauses are evaluated -- this means that when you give a command string as an argument to a command (eg, macro , while , etc) it must be enclosed in quotes (or braces) to stop it being executed too early.
eg,

```
    (macro 'name'
    {
  (somecommand)
    })
```

NOT,

```
    (macro 'name'
  (somecommand)
    )
```

this would assign the value of (somecommand) to the macro "name" not the actual command string itself.

The text enclosed by the quotes, or braces, may enclude any of the standard escape sequences supported by JEd.

See also, Escape Sequences , String s

1.16 Number Clause

Number Clause.

Number clauses always produce a value of type number .

The syntax is,

```
    0xhex_number
    0octal_number
    decimal_number
```

Each variation may optionally have a minus sign '-' preceding it.

1.17 Character Clause

```
                Character Clause.
-----------------

Value is of type
                number
                .

Syntax is,

    ~c

Value produced is ascii value of character ’c’, ie, ~a would produce a value
of 97.
                Escape sequences
                 may be used instead of a character.

    ^c

Value is control-’c’
```

1.18 Null Clause

```
Null Clause.
------------

The null clause has no value, it is mainly used to create variables without
giving them a value.

Syntax is
    @
```

1.19 Symbol

```
                Symbols.
========

JEd maintains one large symbol table which contains all global symbols, this
includes,
    commands
    variables
    macros

Each command string interpreted is also given a symbol table (actually a
list :) which contains symbols local to that string. Local symbols can
contain the same thing as global symbols.

The idea is that you can reference any local symbol which is either on the
same depth of recursion as where it is being referenced from or on a
previous level of recursion.
```

```
When a local variable goes out of scope (when the command string it was
declared in has been left) it will be automatically removed.

All symbols are case sensitive.

See also,

                Symbol Clause
                ,
                addsym
                ,
                remsym
                ,
                global
                ,
                local
                ,
                macro
                ,
                symboldump
```

1.20 Escape Sequences

```
Escape Sequences.
=================

An escape sequence is introduced by the backslash '\' character, the
supported sequences are,

    \n       insert a newline character
    \t       insert a tab character
    \f       insert a form feed character
    \0xFF   insert a hex byte
    \0377   insert an octal byte
    \255    insert a decimal byte

Any other character after the backslash is just copied into the text (or
whatever). So to have a literal '\' character in a string you would need the
sequence '\'. This feature can also be used to suppress clause-inducing
characters such as, quotes, parentheses, braces, @, ^, etc...
```

1.21 Keyboard Mappings

```
                Keyboard Mappings.
******************

This is what the startup script in macros/ binds to each key.

    esc     prompt for
                command string
                 and execute it
    help
```

```
            sleep
            /
            unsleep
             window
ctrl
            close
             window

up      up one line
down    down one line
left    left one column
right   right one column
shift up    up one page
shift down    down one page
shift left    to sol
shift right   to eol
ctrl up   first line
ctrl down   last line
ctrl left   previous word
ctrl right    next word

tab     next tab stop
shift tab   previous tab stop
ctrl tab    insert a tab
return    split line
backspace
            delete
             char behind cursor
delete    delete char under cursor
shift backspace delete to sol
shift delete  delete to eol
ctrl delete   delete line
ctrl backspace  delete word

ctrl b    toggle
            block
             marks
alt b   set start of block
alt B   set end of block
ctrl alt b    clear block
ctrl i
            insert
             block
ctrl z
            delete
             block
ctrl x
            cut
             block to clipboard unit 0
ctrl c
            copy
             block to unit 0
ctrl v    insert clipboard unit 0
ctrl q    delete to end of line
ctrl y    delete all of line
ctrl u    undelete line (from ctrl q/y)
ctrl l
            undo
             line (only if cursor is on correct line)
ctrl L    always undo line

ctrl o
            open file
             from string prompt
ctrl alt o    open file from file req.
ctrl I    insert file from prompt
ctrl alt I    insert file from file req.
ctrl O
            open file in new window
             from prompt
ctrl alt O    open file in new window from file req.
ctrl n
            open new view
             of this file
ctrl w
            save file
             to where it was loaded from
ctrl W    save file as result of string prompt
ctrl alt W    save file as result of file req.
ctrl N
            rename
             this file
ctrl d    change current directory
ctrl k
            clear
             file

ctrl F    set
            find
             string, and find next occurrence
ctrl f    find next occurrence
ctrl alt f    find previous occurrence
ctrl R    set
            replace
             string
ctrl r    replace and find next
ctrl g    find
            reference
             for word under cursor
ctrl G    find reference for specified word
ctrl h    find matching bracket
ctrl j    jump to a line

ctrl ,    activate next window
ctrl .    activate previous window

ctrl s    execute
            script
             file
ctrl alt s    execute the current line
ctrl S    execute the marked block
ctrl alt S    execute the whole of the current file
```

```
    f1      move to bookmark 1
    f2      move to bookmark 2
    f3      move to bookmark 3
    f4      move to auto-mark
    shift f1    set bookmark 1
    shift f2    set bookmark 2
    shift f3    set bookmark 3

    alt d   insert current date
    ctrl e    prompt for AmigaDOS commandstring and execute it
```

1.22 ARexx

```
                ARexx.
******

All copies of JEd run will try to create an ARexx message port, the first
will be called 'JED.1', subsequent ports will be 'JED.2', 'JED.3'...


                Command string
                s can be sent to JEd, they will probably need to be enclosed
in quotes so ARexx doesn't try to interpret them.

The way that results are returned to ARexx is slightly different to most
ARexx supporting applications, successful commands return 1 not zero in the
RC variable. If the result of a command is a string RC will be zero and the
RESULT variable will contain the string.

ARexx macros can be implicitly invoked simply by specifying their name in a

                symbol clause
                , any arguments given to the
                clause
                 will be resolved into
strings and passed to the macro as its arguments.
So, if you had a REXX macro called "foo" and it wanted an argument of "bar"
you could execute it with the
                command string
                ,

    (foo `bar')

If you want to start a macro in this way it _MUST_ reside in the REXX:
assignment.
Also, the value of the clause which implicitly calls a REXX macro will only
represent whether or not it was possible to _start_ the macro running, see
the next paragraph...

Currently there's no way to receive a result from an ARexx macro. This fact
probably won't change since they have to be run asynchronously with the
editor.

See also,
```

rexx

1.23 Title Bar

Title Bar.
****

The title bar of a window is used to display some useful information about
the file being edited in this window. It will be something like,

```
          Word-wrap____     __Block is fully marked
           \   /
filename+ (col,line) total_lines line(s) AWNBbx  <-- `savetabs' setting
  |           __/  |  \__
when present denotes     Auto-indent     |   Block is partially marked
that file contains           Window position won't
unsaved changes.           be saved on exit
```

eg,
jed.doc+ (11,1318) 1383 line(s) AN2

The title bar is also used to display messages (use the
settitle
command to
do this).

1.24 Miscellaneous Notes

Miscellaneous Notes.
****

The maximum length of any line is 32768 characters, there are no problems
loading lines this long either (anymore). The maximum number of lines you
can have is 2147483648. I think that these limits won't be too restrictive.
Currently no checking is done to make sure that these limits aren't broken,
this means that you can crash the system if you do. (Actually if these are
too restrictive it would be relatively easy to double them.)

Sometimes error messages will be shown (on the titlebar) which may seem to
be a bit strange. These will normally be of the type "syntax error: argument
n should have been a xxx" and they are normally encountered when you cancel
a requester or prompt (or when some command types fail). These just show
that the command that wanted the input you didn't give is complaining at
being given nothing (huh?).

JEd appears to be mungwall-clean and to not permanently steal any resources,
I haven't been able to run it under Enforcer (no mmu!), if any hits are
found please send me the output together with information as to the version
of jed you're using and how to recreate the hit.

The prompt mechanism used by the commands
cli

```
                ,
                getstr
                 and
                getnum
                 responds to
these keypresses,
    return  --  accepts the string
    esc     --  cancel the prompt
    bs      --  delete the character behind the cursor
    up/down --  recall the string entered in the last prompt
    (any other keys are just inserted into the string)

Any of the executable files may be made resident (they are pure).

If you find that the editor just exits back to the CLI, with no error
messages when executed, it means that the required disk-based libraries
aren't available or that there is insufficient memory.

It is now possible to mark blocks with the mouse, every time the left mouse
button is double-clicked the command,
    (
                block
                 `t')
is executed.
```

1.25 MakeRefs

```
                MakeRefs.
*********

usage:
    makerefs [-new] [-full] <reffile> {<files>}

    -new  create <reffile>, don't append to it
    -full write fully qualified filenames to the <reffile>, not
    relative to the current directory.
    <reffile> the file to write the index in
    {<files>} files to scan for references, standard AmigaDOS
    wildcards are acceptable.

This program calculates the reference indexes for use with JEd 2.x. The type
of reference scanned for depends on the suffix of each file, there are three
methods,

    1) Files which end in .h This is very poor, all structure definitions
    which have "struct " in the first column of the file will be referenced.
    The reference created will load the whole header file and put the cursor
    on the first line of the structure definition. Hopefully this will
    enable you to reference all system structures (actually the only
    structure it can't handle is ExtendedNode in graphics/gfxnodes.h,
    spurious references are generated since it's formatted strangely). You
    shouldn't attempt to reference the *_protos.h files. So to reference all
    include files cd to the directory holding them and type,

  1> makerefs .jrefs ~(clib)/#?.h
```

```
    then put your include file directory in the path jed searches with the

                addpath
                 command.


    2) Files which end in .c These are assumed to be C source code files,
    all function definitions are referenced if they are in this format,

    rtn-type
    funcname(args...)
    {
    ...code

    3) Any other files These are scanned for autodoc style sections of text,
    ie, things like,

    a.library/AFunction            a.library/AFunction

    This will produce a reference for AFunction.

See also,

                addpath
                ,
                getref
                ,
                rempath
                .
```

1.26 Command Index

```
                Command Index.
**************


                +
                                   addition

                -
                                   subtraction

                *
                                   multiplication

                /
                                   division

                %
                                   modulus

                <<
                                  left shift
```

>>	right shift
~	bitwise NOT
!	logical NOT
|	bitwise OR
||	logical OR
&	bitwise AND
&&	logical AND
^	bitwise EOR
^^	logical EOR
=	set value of a variable
==	test for equality
!=	test for inequality
>	greater than
<	less than
>=	greater than or equal to
>=	less than or equal to
activatefile	activate a named file
addpath	add a directory to the path searched for references
addsym	make a new global symbol
arg	get argument to macro

atol	convert ascii string to number
bind	bind a command string to a keypress
block	control block marks
break	break out of command strings
car	extract first item in list
cd	change current directory
cdr	extract all but first item in a list
changecase	toggle case of some characters in the file
changes	set change counter
clear	reset file
cli	prompt for command string, then execute it
close	close window
copy	copy some of file to clipboard
cut	cut some of file to clipboard
delete	delete some of file
dlock	forbid/permit window refreshing
dowhile	control structure
export	increase the scope of local symbols
extract	get some text from the file

find	find a string
format	'printf' style string formatting
freq	file requester
getnum	request number
getpref	get value of a preference option
getref	load reference description
getstr	prompt for a string
global	create a new global variable
if	control structure
ilock	input lock
info	get information about stuff
insert	insert some text
isalpha	test for an alphabetic character
isalnum	test for alphnumerical character
isdigit	test for a numerical character
isspace	test for whitespace character
join	join two lines
local	create a variable local to this macro
macro	define a macro (subroutine)

match	wildcard string comparer
menu	menu on/off
move	move cursor
nargs	number of arguments passed to macro
newfile	open a new file in a new window
newview	open a new view of this file
nextwind	activate next window
nop	nothing
openfile	open a new file in this window
poke	put character into cursor position
position	change window dimensions/position
prevwind	activate previous window
remsym	remove global symbol
rempath	remove reference path
rename	rename file
renamesym	rename global symbol
replace	replace string found by 'find'
req	requester
return	return value from macro/command string

```
rexx
                send command to ARexx

savefile
            save file to where it was loaded from

savefileas
          save file to specified file

saveprefs
           set whether preferences will be saved on exit

savesection
         save part of file

script
              execute script file

select
              control structure

setmenu
             create menubar

setpref
             set a preference option

settitle
            set title-bar

sleep
               iconify window

split
               split line at cursor

substr
              extract string from another string

symboldump
          dump contents of symbol tables

system
              execute AmigaDOS command

tolower
             make some text lower case

toupper
             make some text upper case

type
                find the type of a value

unbind
              remove command string from keypress
```

undo
undo changes to current line

unsleep
un-iconify window

while
control structure

See also,

Syntax Definitions Explained
,
Section Type Definitions
,
Command Groups
.

1.27 Command Groups

Command Groups.
****

Window commands,

activatefile
,
close
,
menu
,
newfile
,
newview
,
nextwind
,
position
,

prevwind
,
settitle
,
sleep
,
unsleep
,
setmenu
.

File commands,

activatefile

,
cd
,
changes
,
clear
,
newfile
,
openfile
,
savefile
,

savefileas
,
savesection
.

Text manipulation,

block
,
changecase
,
copy
,
cut
,
delete
,
extract
,
find
,
insert
,
join
,
move
,

poke
,
replace
,
split
,
tolower
,
toupper
,
undo
.

Configuration,

bind
,
getpref
,
macro
,
menu
,
saveprefs
,
setmenu
,
setpref
,
unbind
.

Programming,

+
,
-
,
*
,
/
,
%
,
<<
,
>>
,
~
,
!
,
|
, ||,
&
,
&&
,
^
,
^^
,
=
,
==
,
!=
,
>
,
<
,

```
>=
,

<=
,
addsym
,
arg
,
atol
,
break
,
car
,
cdr
,
cli
,
dowhile
,
dlock
,
export
,

format
,
freq
,
getstr
,
getnum
,
global
,
if
,
ilock
,
info
,
isalpha
,
isalnum
,

isdigit
,
isspace
,
local
,
macro
,
match
,
                nargs
                ,
                nop
                ,
                req
                ,
                remsym
                ,

                renamesym
                ,
                rexx
                ,
                return
                ,
                select
                ,
                settitle
                ,
                script
                ,
                substr
                ,
                symboldump
                ,

                system
                ,
                type
                ,
                while
                .

Referencing,

                addpath
                ,
                getref
                ,
                rempath
                ,

See also,

                Syntax Definitions Explained
                ,
                Section Type Definitions
                ,
                Command Index
```

1.28 Syntax Definitions Explained

```
Syntax Definitions Explained.
=============================
explanation of syntax definitions in command reference pages:

    (command arg1  arg2 ...)
rtn      arg1  arg2
type       type  type

The rtn type and the arg type show the kind of values the command returns
and expects to be given, they can be one of the following,

    ()       --  anything (can be void)
    (S)      --  string value
    (N)      --  numeric value
    (S|N)    --  string or numeric value

Arguments surrounded by <...> are compulsory and must be provided for the
command to work, arguments surrounded by [...] are optional and arguments
surrounded by {...} mean one or more arguments can be given.

If I have shown that a command returns a number but have not documented what
it will be, then this scheme will apply, a zero means that the command
failed. If the return is non-zero (usually 1) the command was successful.

Another convention which I have used is that if a command is passed an
incorrect type of value (ie, a number instead of a string, or nothing at
all) the command will not return _any_ value. This will in turn make any
command using the value of this command as an argument fail, and so on...
```

1.29 Section Type Definitions

```
                Section Type Definitions.
=========================

Many commands which deal with parts of the text file expect what I have
referred to as a section type, often this is the argument <section>, this
should be one of the following strings,

    c      --  character under the cursor
    p      --  the character behind the cursor (previous)
    n      --  the character after the cursor
    w      --  the word under the cursor (alpha-numeric only)
    b      --  the currently marked block (the block will then be unmarked)
    l      --  the whole line that the cursor is on
    f      --  the whole file
    sf       --  from the cursor to the start of the file
    sl       --  from the cursor to the start of the line
    ef       --  from the cursor to the end of the file
    el       --  from the cursor to the end of the line
    mX       --  from the cursor to bookmark number X (ie, ‘m1’)

eg, to copy a marked block,
    (
                copy
                 ‘b’ 0)
```

1.30 +

```
   (+ <value1> <value2>)
(N)     (N)   (N)
```

Returns <value1> + <value2>.

1.31 -

```
    (- <value1> [value2])
(N)     (N)   (N)
```

Returns <value1> - [value2]. If no [value2] is provided then <value1> is negated and returned.

1.32 *

```
    (* <value1> <value2>)
(N)     (N)   (N)
```

Returns <value1> * <value2>.

1.33 /

```
    (/ <value1> <value2>)
(N)     (N)   (N)
```

Returns the quotient from <value1> / <value2>.

1.34 %

```
    (% <value1> <value2>)
(N)     (N)   (N)
```

Returns the remainder from <value1> / <value2>.

1.35 <<

```
    (<< <value> <count>)
(N) (N) (N)
```

Returns the <value> left-shifted <count> bits.

1.36 >>

```
    (>> <value> <count>
(N) (N) (N)
```

Returns the <value> right-shifted <count> bits.

1.37 ~

```
    (~ <value>)
(N)    (N)
```

Returns the bitwise NOT of <value>.

1.38 !

```
    (! <value>)
(N)    (N)
```

Returns the logical NOT of <value>. ie, not_zero <=> zero.

1.39 |

```
    (| <value1> <value2>)
(N)    (N)   (N)
```

Returns the bitwise OR of <value1> and <value2>.

1.40 !

```
    (! <value>)
(N)    (N)
```

Returns the logical NOT of <value>. ie, not_zero <=> zero.

1.41 &

```
    (& <value1> <value2>)
(N)    (N)   (N)
```

Returns the bitwise AND of <value1> and <value2>.

1.42 &&

```
    (&& <value1> <value2>)
(N) (N)   (N)
```

Returns the logical AND of <value1> and <value2>.

1.43 ^

```
    (^ <value1> <value2>)
(N)     (N)   (N)
```

Returns the bitwise EOR of <value1> and <value2>.

1.44 ^^

```
    (^^ <value1> <value2>)
(N) (N)   (N)
```

Returns the logical EOR of <value1> and <value2>.

1.45 =

```
                    (= <name> <value>)
(N)     (S)     (S|N)
```

Sets the contents of the variable <name> to <value>. Both
global
and
local
variables may be set, but the variable must already have been created.

See also,

addsym
,
global
,
local

1.46 ==

```
    (== <value1> <value2>)
(N) (S|N)   (S|N)
```

Returns 1 if <value1> is equivalent to <value2>. Strings are compared case insignificantly.

1.47 !=

```
    (!= <value1> <value2>)
(N) (S|N)   (S|N)
```

Returns 1 if <value1> is not equivalent to <value2>. Strings are compared case insignificantly.

1.48 >

```
    (> <value1> <value2>)
(N)     (N)   (N)
```

Returns 1 if <value1> is greater than <value2>.

1.49 <

```
    (< <value1> <value2>)
(N)     (N)   (N)
```

Returns 1 if <value1> is less than <value2>.

1.50 >=

```
    (>= <value1> <value2>)
(N) (N)   (N)
```

Returns 1 if <value1> is greater than or equal to <value2>.

1.51 <=

```
    (<= <value1> <value2>)
(N) (N)   (N)
```

Returns 1 if <value1> is less than or equal to <value2>.

1.52 activatefile

```
                     (activatefile <file>)
(N)      (S)
```

Attempts to make a window holding <file> the active window, if <file> is not already in memory an attempt will be made to load it into a new window.

```
See also,

                openfile
```

1.53 addsym

```
                    (addsym {<name> <value> <sym-type>})
(N)      (S)     ()      (N)

Creates a new
                symbol
                 called <name> with a value of <value>.

The <sym-type> argument determines whether the symbol is global or local,
and whether it is treated as a command or as a variable. <sym-type> can be,

    1/STF_GCOM   global command
    2/STF_GVAR   global variable
    3/STF_LCOM   local command
    4/STF_LVAR   local variable

The variable types always return their _literal_ value when accessed,
command types return their interpreted value, for example,

    (addsym
  `foo'   `foo'   STF_LVAR
  `bar'   {`bar'} STF_LCOM
    )
    (settitle (format `foo = %s, bar = %s' foo bar))

This creates two local symbols, foo and bar, foo is a variable string and
bar is a command string (ie, it will be interpreted), then displays their
values in the title bar of the window. This is a pathetic example.

See also,

                Symbol
                ,
                global
                ,
                local
                ,
                remsym
                ,
                renamesym
                ,
                =
                ,
                export
```

1.54 addpath

```
                    (addpath {<dir>})
(N)
```

Adds a directory to the list of directoried scanned for reference indexes by the getref command.

See also,

MakeRefs , rempath , getref

1.55 arg

```
                    (arg <index> <type> <prompt>)
(S|N)  (N)   (S)  (S)
```

MACRO-ONLY.

Returns the <index>'th argument passed to the macro on invocation. If no argument was supplied it is prompted for with the string <prompt>. If the argument is not of the type specified by <type> (s = string, n = number, e = either) the macro will be automatically aborted.

See also,

nargs , macro

1.56 atol

```
    (atol <string>)
(N)   (S)
```

Returns the number represented by the ascii <string>. Decimal, hex and octal bases are supported.

1.57 bind

```
                    (bind {<key> <command>})
(N)    (S)   (S)
```

Binds the <command> string to <key>. Remember that the command string must be enclosed by quotes or braces.

```
<key> should be a string containing any number of qualifiers then one key.
The recognized words are,

    qualifiers
  SHIFT
  ALT
  CONTROL/CTRL
  COMMAND/AMIGA
  NUMERICPAD
  LMB     -- left mouse button
  MMB     -- middle mouse button
  RMB     -- right mb (currently unuseable)

    keys
  SPACE
  BACKSPACE
  TAB
  ENTER
  RETURN
  ESC/ESCAPE
  DEL/DELETE
  HELP
  UP
  DOWN
  RIGHT
  LEFT
  F1 ... F10
  and usual ascii characters (a,b,...)

some example commands

    (bind `shift tab' {(move `lt' 1)})

    (bind
  `j'                 {(req `hello' `world')}
  `lmb numericpad *'  {(settitle `foo')}
    )

If you bind onto a key which already has a binding the old command string
will not be lost, if you subsequently
                unbind
                 the key the old binding will
come back into effect.

See also,

                Command String
                s,
                unbind
```

1.58 block

```
                    (block <type>)
(N)     (S)
```

```
Set the block markings according to <type>, this is a standard
                section type
                or,
    s --  mark start of block
    e --  mark end of block
    k --  kill both block marks

    t --  cycle through the above options

See also,

                Section Type
                ,
                copy
                ,
                cut
                ,
                insert
```

1.59 break

```
                    (break <depth>)
()      (N)

Stops the execution of <depth> number of strings, execution will continue
with the next clause in the <depth> - 1 previous string.

In the following example the (break) will cause a branch to the
                req
                 command
displaying the <depth> broken.

    (
                if
                 1
    {
  (if 1
  {
      (if 1
      {
    (break 2)
    (
                req
                 `0' `zero')
      })
      (req `1' `one')
  })
  (req `2' `two')
    })
    (req `3' `three')

eg, if you change the '(break 2)' to '(break 1)' only the '(req `0'...)'
will be skipped. Test it out.
```

See also,

dowhile
,
if
, seclect,
while
,

1.60 car

```
                    (car <list> <sep-char>)
(S)  (S)  (N)
```

This command is used (in conjunction with
cdr
) to manipulate lists of words
separated by a single character, <sep-char>.

It returns the first item in <list>, ie, if you executed the command,

```
    (car 'one,two,three' ~,)
```

you would get a value of,

```
    one
```

If <sep-char> does not occur in the list the whole <list> is returned.

See also,

cdr

1.61 cd

```
    (cd <dir>)
(N) (S)
```

Makes <dir> the current directory for the editor.

1.62 cdr

```
                    (cdr <list> <sep-char>)
(S)  (S)  (N)
```

This command is used (in conjunction with
car
) to manipulate lists of words
separated by a single character, <sep-char>.

It removes the first item in the list and returns the remainder (without the leading <sep-char>).

For example if you executed,

```
    (cdr ‘one,two,three’ ~,)
```

you would get a value of,

```
    two,three
```

If <sep-char> does not occur in the <list> a null string ("") is returned.

See also,

car

1.63 changes

```
    (changes <number>)
(N)      (N)
```

Sets the counter of changes to the current file to <number>.

1.64 cli

```
                    (cli)
()
```

Prompts for a command string and then executes it. Note that as with all commands who use the prompt mechanism a sleeping window will be woken up.

Returns the value of the executed command.

This command is equivalent to (script ‘s’ (getstr ‘cmd> ’))

See also,

Command String , getstr , script

1.65 close

```
                    (close)
(N)
```

Closes the current window, if it is the only view of the file the file will be unloaded. If it is the last window that the editor has open the present command string will be terminated and everything will exit.

If using this from a script do NOT assume which window will be activated when this one closes. It is left up to Intuition to decide which window to activate. Until it does this (it may not even activate one of my windows, or if it does I won’t hear about it until after processing the script) the window which the editor reguards as ’active’ is guaranteed to be a view of the file which closed (if there are any other views).

Be warned, this command is weird.

See also,

newfile
,
newview
,

1.66 copy

```
                    (copy <section> <unit>)
(N)    (S)      (N)
```

Copies a section of text to the clipboard device. <unit> is the clipboard unit to copy to (usually 0). A <unit> of -1 means copy the text to my internal clipboard unit, this can be useful for copying between windows of the editor.

See also,

Section Type
s,
cut
,
insert

1.67 cut

```
                    (cut <section> <unit>)
(N)   (S)      (N)
```

The same as
copy
except that the section of text copied is then deleted from the file.

```
See also,

                Section Type
                s,
                copy
                ,
                delete
                ,
                insert
```

1.68 clear

```
    (clear)
(N)

Clears everything to do with the current file, resetting its name to
"Untitled" as well.
```

1.69 changecase

```
                    (changecase <section>)
(N)   (S)

Toggles the case of all alphabetic characters in <section>.

See also,

                Section Type
                s,
                tolower
                ,
                toupper
```

1.70 delete

```
                    (delete <section>
(N)     (S)

Deletes <section> from the file.

See also,

                Section Type
                s,
                cut
```

1.71 dowhile

```
                    (dowhile <body> <condition>)
(N)      (S)    (S)
```

First executes the command string <body>, then executes command string <condition>, if the result of <cond> is non-zero the loop is repeated. This command has the same safeguards against infinite loops as while has.

Note that the <body> and <condition> are in the opposite order than in the while command.

See also,

Command Strings,
while

1.72 dlock

```
                    (dlock <status>)
(N)    (N)
```

Sets the <status> of the display lock. When it is non-zero no rendering is done in the current window (except for on the title bar) The intelligent use of this command can significantly speed up macros.

There could be problems if a macro who has turned on the display lock is aborted, by not being given the correct arguments perhaps, leaving the display locked. If this happens get into the (cli) command and unlock the display.

This command does NOT nest (yet). Each window has its own, independant, lock.

When it is unlocked any queued refreshes are done.

See also,

ilock

1.73 export

```
                    (export {<symbol> <how-far>})
(N)     (S)       (N)
```

This command increases the scope of <symbol> so that <how-far> more command strings can access it than before, the best way to explain this is with an

```
example, (the 'local' macro),

    ; create a macro called 'local'
    ;
    (macro `local' {

  ; local symbol to count the number of arguments we've done
  ;
  (addsym `__i' 0 STF_LVAR)

  ; while we've got more arguments to do...
  ;
  (while {(>= (- nargs __i) 2)} {

      ; create a new local symbol...
      ;
      (addsym (arg (+ __i 1) `s') (arg (+ __i 2) `e') STF_LVAR)

      ; and export it to the command string the macro was called from,
      ; 2 strings "behind" (one for the body of this while loop and
      ; one for the base level of the macro definition)
      ;
      (export (arg (+ __i 1) `s') 2)

      ; increment argument counter
      ;
      (= `__i' (+ __i 2))
  })
    })
```

See also,

Symbol
s,
addsym
,
local
,
remsym

1.74 extract

```
                    (extract <section>)
(S)      (S)
```

Returns the text from <section>.

See also,

Section Type
s

1.75 find

```
                    (find ‘s’ <string>)
(N)

Sets the string which find will search for.


    (find ‘n’)
(N)

Search for the next occurrence of the string set by (find ‘s’). This command
returns 1 if the string was found.


    (find ‘p’)
(N)

Same as (find ‘n’) but searches backwards.


    (find <switch> <status>)
(N)       (S)       (N)

Defines the behaviour of the find command, these <switch>’es are available,

    c -- case dependant search when <status> is non-zero

    w -- when <status> is non-zero the string set by (find ‘s’) is parsed as
    a standard AmigaDOS 2.0 wildcard. Note that the search only extends to
    the end of each line in turn, and that ’#?’ will probably have to be
    added onto the end of the string to account for characters after the
    pattern that you are searching for.

    r -- enable regular expressions, when this <switch> is on the above two
    switches have no effect. The nearly-public-domain regexp library by
    Henry Spencer is used so refer to that for more details, basically these
    are the meta-characters recognized, (they can be un-recognized by
    backslash-escaping them),

    .       matches any single character

    [abc]   match a, b, or c
    [a-z]   match any character in range from a to z
    [^e]    match any char except e
            the above types of character classes can be combined, so,
                [a-zA-Z_]
            matches any alphabetical chacter or the underscore

    ^       matches the beginning of the line of text being compared
    $       matches the end of the line of text

    a|b     matches either expression a or expression b

    ()      the actual text that is matched by the RE between the
            parentheses is remembered. It can be recalled when substituting
            for an RE with the
                replace
                 command.

    *       matches the preceding expression 0 or more times
    +       matches the preceding expression 1 or more times
    ?       matches the preceding expression 0 or 1 times

    Some examples of regular expressions could be,

    <[a-z]*/([a-z_]*).h>
    this would match "<clib/exec_protos.h>" saving "exec_protos" for recall
    as "\1" by
                replace
                , but would not match "<stdio.h>",
    "<Clib/Exec_protos.h>", etc.

    ^[a-zA-Z_]*\(
    note the escaped parenthese so that it takes its literal value, this
    could match "function(", "Func_tion(" or "(", all beginning at the start
    of a line, but not, "function", etc...

    There is a slightly confusing feature when searching _backwards_ for
    regular expressions, that is that instead of searching from right to
    left in a line it searches left to right (it still goes bottom to top
    though :), I don't believe that this is too much of a problem, just bear
    it in mind.

 See also,

                replace
```

1.76 freq

```
                     (freq <type> <title> <startpos>)
(S)    (S)   (S)    (S)

Opens a file requester and asks for a filename. <type> can be 'r' or 'w',
these stand for read and write. <title> is the title of the requester window
and <startpos> is the file (and dir.) to start the requester from.

If the requester is cancelled no result is returned, this will probably
abort any commands who want it as an argument.

See also,

                getstr
```

1.77 format

```
    (format <fmtstring> {[values]})
(S)      (S)    (S|N)
```

```
Returns a formatted string made from the format specification <fmtstring>
and the [values]. (Almost) standard C language formatting is done, these
substitutions can be performed,

    %s      insert string
    %ld     insert decimal value
    %lx     insert hex value
    %lc     insert char value

eg,
    (format ‘%s %ld’ ‘string’ 1000)
```

1.78 getref

```
                    (getref [refname])
(N)

This command searches all directories in the reference path (set with
(addpath)) for files called ".jrefs", these files should contain indexes to
all available references. If a reference matching refname (or the word under
the cursor if refname isn’t given) is found a new window is opened and the
text for that reference is displayed. For example if you make a reference
file for all autodoc files you can, when programming, place the cursor on a
function name and then bring up the explanation of that function.

Each line in a .jrefs file which begins with a @ character is taken as a
valid reference, there are three types of line format,
  @refname@reffile@searchstring@
    reffile is loaded and searchstring is looked for in the start of each
    line. If found the cursor is set to the start of that line.

  @refname@reffile@#startpos@
    reffile is loaded and the cursor is moved to startpos (a decimal number)
    many bytes into the file.

  @refname@reffile@#startpos/#endpos@
    the section of text between startpos and endpos (both decimal offsets)
    is loaded into the window.

  @refname@reffile@^startline@
    cursor is positioned at startline (a decimal number).

In each case refname is the name of the reference, this is matched
case-significantly with what is being searched for. reffile is loaded
relative to the directory that the .jrefs file containing it is found in.

The program makerefs is provided for making references for autodocs, C
header files and C source files, see the file doc/makerefs.doc

See also,

                MakeRefs
                ,
                addpath
```

,
rempath

1.79 getstr

```
                    (getstr <prompt>)
(S)     (S)
```

Prompts the user for a string, if the prompt is cancelled (<esc>) no value will be returned.

See also,

getnum

1.80 getnum

```
                    (getnum <prompt>)
(N)     (S)
```

Prompts the user for a numeric value, if the prompt is cancelled no value is returned.

See also,

getstr

1.81 getpref

```
                    (getpref <pref>)
(N|S)      (S)
```

Returns the current setting of preference option <pref>. Currently you can't get the font settings.

See also,

setpref

1.82 global

```
                    (global {<name> <value>})
(N)      (S)     ()
```

Creates a new global variable <name>, its value will be set to <value>.

This command is actually implemented as a macro, in the startup file.

See also,

addsym
,
local
,
remsym
,
=

1.83 if

```
                    (if <condition> [true-cmd] [false-cmd])
()   (N)      (S)         (S)
```

If <condition> is non-zero the
command string
[true-cmd] is executed, else,
[false-cmd] is executed. The result of this command is the result of the
string executed, or no value if the string which should have been executed
wasn't provided.

See also,

dowhile
,
select
,
while

1.84 ilock

```
                    (ilock <status>)
(N)    (N)
```

Sets the status of the input lock. This is intended for use by
ARexx
macros
to lock out user input. Only commands from ARexx are received. Input through
the window just queues up until the <status> is set back to zero. (actually
the
getstr
and
getnum
commands are allowed to break the lock). The returned
value is the OLD status of the lock.

It is polite behaviour to reset the lock to whatever it was before you set
it, ie, from ARexx,

```
    '(ilock 1)'
    oldilock = rc
    ...your code...
    '(ilock 'oldilock')'
```

Unlike the
dlock
command this lock is global ie, it affects everywhere, you can't just lock the input from one window.

See also,

ARexx
,
rexx
,
dlock

1.85 insert

```
                    (insert <section>)
(N)     (S)
```

Inserts section into the file at the current cursor position, since you can't insert into the text to be inserted it is probable that only blocks can be inserted with this command.

```
    (insert `f' <file>)
(N)   (S)
```

Inserts the file <file>.

```
    (insert `s' <string>)
(N)   (S)
```

Inserts the string <string>.

```
    (insert `a' <value>)
(N)   (N)
```

Inserts the ascii code <value> into the file.

See also,

Section Type
s,
copy
,
cut

1.86 info

```
    (info <type>)
(S|N)    (S)
```

Returns some information about the editor and its current environment.
<type> can be,

```
    col   column number (N)
    cols  number of columns in this line (N)
    line  line number (N)
    lines number of lines in this file
    char  ascii value of character under cursor (N)
    views number of views open of this file (N)
    files number of separate files open (N)
    windows total number of open windows (N)
    time  current time "HH:MM:SS:" (S)
    date  todays date "DD-MMM-YY" (S)
    cd    current directory (S)
    fullname  fullname of current file (includes path) (S)
    filename  basename of current file (S)
    dirname path of current file’s directory (S)
    screenx width of screen (N)
    screeny height of screen (N)
    leftedge  x position of window (N)
    topedge y position of window (N)
    width width of window (pixels) (N)
    height  height of window (N)
    size  number of characters in file (no tab optimization) (N)
    offset  distance from start of file (1st char = 1) (N)
    asleep  1 if window is sleeping (N)
    port  name of ARexx port (S)
    rev   release number (N)
    barheight height of title bar (N)
```

1.87 isalpha

```
                    (isalpha <char>)
(N)      (N)
```

Returns non-zero if <char> is a member of the alphabet.

See also,

isalnum

isdigit

1.88 isalnum

```
                    (isalnum <char>)
(N)      (N)
```

```
Non-zero if <char> is alphabetic or numeric.

See also,

                isalpha
                ,
                isdigit
```

1.89 isdigit

```
                    (isdigit <char>)
(N)      (N)

Non-zero if <char> is a digit.

See also,

                isalpha
                ,
                isalnum
```

1.90 isspace

```
                    (isspace <char>)
(N)      (N)

Non-zero if <char> is a white space character.

See also,

                isalpha
                ,
                isalnum
```

1.91 join

```
                    (join)
(N)

Joins this line to the following one, if there is no line below this one it
has no effect.

See also,

                split
```

1.92 local

```
                    (local {<name> <value>})
(N)     (S)     ()
```

Creates a variable local to this command string and any command strings entered from this one, when this command string is exited all local symbols associated with it are discarded.

The variable will contain <value>.

Local variables always take precedence over global variables of the same name.

Note: This command is actually a macro in the startup file.

See also,

addsym
,
macro
,
=
,
export

1.93 macro

```
                    (macro <name> <commands>)
(N)    (S)    (S)
```

Creates a macro symbol of <name> with an associated command string of <commands>.

Macros are treated by jed (almost) exactly the same as normal (primitive) commands, they are also kept in the same hash table so you could redefine a primitive command as a macro :-). Macros are invoked in the same way as commands and can have arguments given to them (through the arg command) and return a value (return command).

An example macro could be,

```
    ; Word count macro.
    (macro ‘wc’
    {
  (dlock 1)         ; lock display update
  (local ‘words’ 0)                  ; word counter
  (move ‘sm’ 0)                      ; save our position
  (move `sf')                         ; go to top of file
  (while {(move `nw' 1)}              ; loop till we get to last word
  {
      (= `words' (+ words 1))         ; increment counter
  })
  (req `There are %ld word(s) in this file.' `wow!' words)
  (move `bm' 0)                       ; back to old position
  (dlock 0)         ; unlock display
  (return words)          ; return the number of words
    })
    ; To invoke this macro type (wc) at the command line
    ; prompt (normally <ESC>)
```

```
Note: This command is actually a macro definition in the startup file, yes
thats correct, the macro command is a macro ;)

See also,

                Command String
                s,
                Symbol
                s,
                addsym

                arg
                ,
                local
                ,
                nargs
                ,
                return
                ,
                renamesym
```

1.94 move

```
    (move <type> <number>)
(N)    (S)   (N)

Moves the cursor according to <type>, which can be,

    d --  move down <number> lines
    dp  --  move down <number> pages
    u --  move up <number> lines
    up  --  move up <number> pages
    ln  --  move to line <number>
    cn  --  move to column <number>
    nc  --  move <number> characters ahead
    nw  --  move <number> of words ahead
    pc  --  move <number> of characters back
    pw  --  move <number> of words back
    r --  move <number> columns right
    rt  --  move <number> of tabs right
    l --  move <number> columns left
    lt  --  move <number> of tabs left
    of  --  move <number> characters from sof

    bm  --  move to bookmark <number>
    sm  --  set bookmark <number>, there are 65535 bookmarks from -32767
      through 0 to +32767. Bookmarks track any changes to the file
      and are cleared when a new file is started. Bookmarks are shared
      between all views of a file.
```

The difference between ‘nc’ and ‘r’ is that nc will move onto the start of the next line at the eol whereas r will just keep moving right (to a maximum of 32768 columns!).

```
    (move <type>)
(N)   (S)
```

There also these move commands which don’t take a <number> argument,

```
    ef  --  move to the last line
    el  --  move to the last column
    sf  --  move to the first line
    sl  --  move to the first column
    bs  --  move to block start
    be  --  move to block end
    am  --  move to the auto bookmark, this is set after a large(ish) move
      command, or after the find command.
    mb  --  move to the next bracket which is at the same level of nesting
      as the one under the cursor, this is what matches what,
    (   )
    {   }
    [   ]
    <   >
    `   '
```

If the specified position can’t be moved to the command will return 0, otherwise 1.

1.95 match

```
                    (match <pattern> <string>)
(N)    (S)       (S)
```

Matches the AmigaDOS wildcard string <pattern> case-insignificantly with <string> returning 1 if they are equivalent, 0 if they aren’t.

See also,

```
                ==
```

1.96 menu

```
                    (menu <status>)
(N)    (N)
```

Sets whether or not a menu is displayed in this window. If <status> is non-zero the menu is on.

Currently this (probably) has a bug, in that when a window is slept the menu status is not remembered for when it is un-slept.

Note that
setmenu
must have been successfully called for a menu to be displayed.

See also,

setmenu

1.97 nargs

```
                    (nargs)
(N)
```

MACRO-ONLY.

Returns the number of arguments passed to a macro when it was invoked.

See also,

macro
,
arg

1.98 newfile

```
                    (newfile <file>)
(N)      (S)
```

Opens a new window for <file>, if <file> exists it will be loaded into the window.

See also,

activatefile
,
close
,
newview
,
openfile

1.99 newview

```
                    (newview)
(N)

Opens an additional window for editing the current file in. The windows
share the same text buffer and bookmarks but are otherwise independant.

See also,

                close
                ,
                newfile
```

1.100 nextwind

```
                    (nextwind <type>)
(N)       (S)

Activates the next window in the list.
<type> can be,
    f --  activate the next _separate_ file
    v --  activate the next view of this file
    a --  step through all windows

See also,

                prevwind
```

1.101 nop

```
    (nop)
(N)

This command does absolutely nothing, it always returns 0.
```

1.102 openfile

```
                    (openfile <file>)
(N)       (S)

Tries to load <file> into the current window (and all other windows of the
same file), if it can't be loaded the window is cleared.

Sequential access files (ie, pipes and serial ports) can also be read (by

                newfile
                 as well). You'll probably need to rename the editor's copy after
reading it in, though.
```

```
See also,

                newfile
```

1.103 prevwind

```
                    (prevwind <type>)
(N)       (S)

Activates the previous window in the list,
<type> can be,
    f --  previous _separate_file
    v --  previous view of this file
    a --  step through all windows

See also,

                nextwind
```

1.104 poke

```
    (poke <char>)
(N)   (N:8)

Sets the current character to <char>, only the lower 8 bits of <char> are
used.
```

1.105 position

```
                    (position <x> <y> <w> <h>)
(N)       (N) (N) (N) (N)

Sets the position of the current window,
    <x> --  x position
    <y> --  y position
    <w> --  width
    <h> --  height

All values are in pixels.

If the window is currently
                sleep
                ing the new position will not be used until
it's woken up.

See also,

                sleep
                ,
```

unsleep

1.106 replace

```
                    (replace 's' <string>)
(N)               (S)
```

Sets the replace string to <string>.

```
    (replace 'r')
(N)
```

If the string under the cursor matches the string set by (find 's') it is replaced with the string set by (replace 's') and the cursor is advanced to the end of the replaced string.

Note that it probably isn't a good idea to replace text found with wildcards.

When regular expression searching has been enabled (see
find
) the actual
piece of text which matched the expression is replaced, the replace string can also include these meta-characters,

```
    &   --  insert the whole of the string which was matched
    \n  --  insert the n'th parenthesized string from the RE (n is a digit
            from 1 to 9).
```

See also,

find

1.107 remsym

```
                    (remsym {<name>})
(N)       (S)
```

Removes the symbol <name>, can be *any* type of symbol whatsoever.

See also,

Symbol
s,
addsym
,
global
,
macro

1.108 rempath

```
                    (rempath {<dir>})
(N)        (S)
```

Removes directories from the reference path list which were added by addpath.

<dir>'s must be exactly the same string as what was given to addpath, it is _not_ enough that the two <dir>'s point to the same place (ie, if the path sys:man is addpath'ed and you try to rempath dh0:man it won't work, even though sys: may well be dh0:).

See also,

addpath
,
getref

1.109 rename

```
                    (rename <name>)
(N)      (S)
```

Change the name of the current file to <name>, this is where the file will be saved to next
savefile
command.

See also,

newfile
,
savefile
,
savefileas
,

1.110 renamesym

```
                    (renamesym {<old-name> <new-name>})
(N)    (S)     (S)
```

This command changes the name of the global
symbol
<old-name> to <new-name>.

The main reason I added this command was to make it very easy to add new features to existing primitive (built-in) commands, for example, if you always want the
openfile
command to use a file-requester if it is not given

a filename you could use this script,

```
    ; rename openfile command to _openfile
    (
                if
                 (renamesym `openfile' `_openfile')
    {
  ; create a macro of name openfile
  (
                macro
                 `openfile'
  {
      ; were we given a filename
      (if (
                ==

                nargs
                 0)
      {
    ; no, so request one and open it
    (_openfile (
                freq
                 `r' `file...' (
                info
                 `fullname')))
      }
      {
    ; yes, so just open it
    (_openfile (
                arg
                 1 `s' `'))
      })
  })
    })
```

See also,

```
                Symbol
                s,
                addsym
                ,
                remsym
                ,
                macro
```

1.111 req

```
                    (req <body> <gads> {[values]})
(N)  (S)  (S) (S|N)
```

Displays a requester containing <body> as its main text and <gads> as its gadgets. The <gads> specification can define multiple gadgets by separating each one by a vertical bar ('|') character.

Both <body> and <gads> can contain format characters (%s, %ld, etc), <body>

```
takes formatting arguments from [values] first. The value returned is the
number of gadget that was selected (starting at 1 for the leftmost gadget)
or zero if the rightmost gadget was selected.

See also,

                format
```

1.112 rexx

```
                    (rexx <type> <string>)
(N)    (S)   (S)

Send a command to ARexx, <type> can be,

    m --  <string> is the name of a macro-file to be executed by ARexx, it
      should have a filename extension of ".jed".
      If you want you can specify any arguments to the macro after
      the macro name, ie,
    (rexx ‘m’ ‘amacro arg1 arg2’)

    s --  <string> is a string of ARexx commands to be executed by REXX.

See also,

                ARexx
```

1.113 return

```
                    (return [result])
()      (S|N)

MACRO-ONLY.

Returns control from a macro to whatever invoked it, the value of the macro
is [result], this can be any type of
                value
                .

This command never returns (for obvious reasons).

See also,

                macro
```

1.114 savefile

```
                    (savefile)
(N)
```

Saves the current file as the file it was loaded from (or as what it has
been
rename
d to).

See also,

openfile
,
rename
,
savefileas
,

1.115 savefileas

```
                    (savefileas <name>)
(N)    (S)
```

Saves the current file as file <name>.

From now on the current file will be called <name> by JEd.

See also,

openfile
,
rename
,
savefile

1.116 savesection

```
                    (savesection <section> <file>)
(N)     (S)      (S)
```

Saves the specified section of text as file <file>.

If you’re writing a macro or ARexx script for something like integrated
compilation this command makes more sense than
savefileas
since it doesn’t
change the state of the ’changes’ counter or the name of the file. ie, use

```
    (savesection ‘f’ ‘t:???’)
```

See also,

savefile
,
savefileas
,

changes

1.117 saveprefs

```
                    (saveprefs <boolean>)
(N)        (N)
```

Allows you to set whether or not options set by the
setpref
command will be
saved into the file "s:jed.config" file when the editor is exited. By default this command is always set to true (ie, non-zero), therefore saving preference files.

Note that contrary to what the name suggests this command doesn't actually save anything, this is only done by the cleanup code.

See also,

Startup
,
setpref

1.118 select

```
                    (select {<condition> <command>} [default-cmd])
()       (N)    (S)       (S)
```

If <condition> is non-zero its corresponding <command> is executed, the result of the executed <command> is returned. If none of the supplied <conditions>'s are non-zero the [default]
command string
is executed (if it
is supplied).

An example,

```
    (select
  (== (info `char') ~a)
  {
      (settitle `a')
  }
  (== (info `char') ~b)
  {
      (settitle `b')
  }
  {
      (settitle `neither')
  }
    )
```

See also,

```
                Command String
                s,
                if
                ,
                dowhile
                ,
                while
```

1.119 setmenu

```
                    (setmenu <file>)
(N)      (S)

Reads <file> and makes a set of menus from it, each line represents one part
of the menu, the format of the different types of lines are,

MENU "name"                     Creates a new menu block

ITEM "name" "key" "commands"    Creates a menu item, key is the command-key
        shortcut, commands is the
                command string
                        which gets executed when the item is selected.

SUB "name" "key" "commands"     Creates a sub item on the last menu item.

BAR       Creates a separator bar in the menu block.

SBAR        Creates a separator bar in the sub item
        block.

END       Terminates the menu definition.

The "key" shortcut may be upper or lower case. Menu shortcuts are only
considered to be case-significant when two shortcuts of the same letter (but
different case) are defined.

See also,

                Command String
                s,
                menu
```

1.120 setpref

```
                    (setpref <pref> [arg1] [arg2])
(N)      (S)

Sets one of the preference settings,

    pref        arg1      arg2
    ----------------------- --------------- ----------------
    tabsize       size (N)
    leftmargin        col (N)
    rightmargin       col (N)
    autoindent        bool (N)
    wordwrap        bool (N)
    font        name (S)      size (N)
    disktab       size (N)
    savetabs        code (N)
    scrollhack        bool (N)
    bakdir        name (S)
    baknum        number (N)
    saveprefs       bool (N)
    nosnapshot        code (N)
    pubscreen       name (S)
```

Explanations:

```
    tabsize
  The size of tabs on the screen.

    disktab
  The size of tabs read from and written to disk.

    savetabs
  Controls whether or not TAB (0x09) bytes are saved in files, code
  can be,
      0 --  no TABs are saved
      1 --  leading spaces in each line are optimized to TABs
      2 --  all spaces which can be are optimized into TABs are and
        trailing whitespace is discarded (except after quotes)

    leftmargin
  Where the cursor is put horizontally after the
              split
               command, unless
  autoindent is on.

    rightmargin
  Where long lines are chopped when wordwrapping.

    font
  The font to use for this window. The <name> must have a ".font"
  suffix.

    bakdir
  The directory which backup files are saved to.

    baknum
  The maximum number of backup copies to keep of each file. Whenever a
  file is saved the previous copy (if it exists) is put into the
  backup directory.
      eg, if bakdir is set to t: and baknum is set to three you would
      get backups like,
    t:afile.bak1  - newest after actual file
    t:afile.bak2
    t:afile.bak3  - oldest
    nosnapshot
  Determines when (or if) the window's dimensions are stored for
  saving to the configuration file. The possible code's are,
      0 --  when the window is closed
      1 --  never
      2 --  now (but never again)

    pubscreen
  Sets the name of the public screen which all new windows are to be
  opened on. The Workbench screen is "Workbench", to use the default
  public screen use a name of "" (a null string).

All the above preferences are local to each window, when a new window is
created it inherits its preferences from its parent. The following
preference settings are global to the whole editor.

    scrollhack
  When on (non-zero) scrolling speed is doubled (ish) when no blocks
  are being displayed. To do this Intuition is fooled into only
  scrolling one bitplane. This option is on by default but you may
  need to turn it off for newer os releases, (those supporting
  AGA???). Thanks to Adriaan van den Brand for this idea.

    saveprefs
  When on (non-zero) the preferences of each file saved will be stored
  in that file's filenote. This option is intelligent enough not to
  overwrite any existing comments which aren't preference
  specifications.

Each file that is loaded will have it's filenote checked to see if it
contains a string that defines some preferences (this is what the saveprefs
preference option does), the string should be formatted like this,

    @@ww<+|->\ai<+|->\ts<n>\dt<n>\st<n>\lm<n>\rm<n>

    ie,
    @@ww-\ai+\ts4\dt8\st2\lm1\rm77

the individual switches are,
    ww  --  wordwrap
    ai  --  autoindent
    ts  --  tabsize
    dt  --  disk tabsize
    st  --  save tabs
    lm  --  left margin
    rm  --  right margin

They can be specified in any order, not all of them have to be given. ie, to
make sure that a particular file _never_ has any tab characters saved in it
you can use the following CLI command,

    1> filenote filename "@@st0"

where filename is whatever the file is called.

See also,
                Startup
                ,

                getpref
                ,

                saveprefs
```

1.121 settitle

```
                    (settitle <title>)
(N)       (S)

Sets the current window’s title string to <string>, this string will remain
in view until the next IDCMP event.

See also,

                Title Bar
```

1.122 script

```
                    (script <section>)
()      (S)

Executes the text in the specified section of the current window.


    (script ‘x’ <file>)
()    (S)

Executes the script file <file>. (‘x’ stands for eXternal). If <file> can
not be found relative to the current directory "s:jed/" will be prepended to
<file> and it will be looked for again.


    (script ‘s’ <string>)
()    (S)

Execute
                command string
                 <string>.

All the script variants return the result of whatever was interpreted, the
text that is scripted doesn’t have to be a
                command clause
                , any
                clause
                -type
is acceptable.

An interesting use of this is to process
```

```
                 escape sequences
                  in a string (from
a prompt), ie, this will prompt for a 'find' string and interpret it to use
any escape sequences,

    (
                 find
                  `s'
  (script `s'
      (
                 format
                  ``%s''
    (
                 getstr
                  `find> ')
      )
  )
    )
```

the format command is necessary to make the string into the correct format for a
string clause
(ie, in quotes).

See also,

Command String
s,
Clause
s,
cli

1.123 sleep

(sleep)
(N)

Make the current window go to 'sleep', it will become a small window on the screen title bar. It can be set back to normal by the (unsleep) command or clicking the right mouse button when the window is active.

All commands (except for those which use the prompt, these will enlarge the window) can be executed while the window is sleeping. Anything you type while a window is asleep will be inserted!

See also,

getstr
,
unsleep

1.124 split

```
                    (split)
(N)

Break the line into two at the cursor.

See also,

                join
```

1.125 substr

```
    (substr <string> <index> <len>)
(S)      (S)       (N)      (N)

Returns a string extracted from <string>, first character is <index>
characters from start of <string>, <len> characters are extracted.
```

1.126 symboldump

```
                    (symboldump <file> <type>)
(N)    (S)     (S)

Writes the symbol table’s contents to <file>, <type> can be,

    globals --  all global symbols
    locals  --  all local symbols
    all     --  all symbols

See also,

                Symbol
                s,
                addsym
```

1.127 system

```
    (system <command>)
(N)      (S)

Executes an AmigaDOS command string. The value of this command is the return
code of the executed command or -1 if the command couldn’t be executed.
```

1.128 toupper

```
                    (toupper <section>)
(N)      (S)

Make all characters in <section> upper case.

See also,
    Section Types,
                changecase
                ,
                tolower
```

1.129 tolower

```
                    (tolower <section>)
(N)      (S)

Make all characters in <section> lower case.

See also,
    Section Types,
                changecase
                ,
                toupper
```

1.130 type

```
                    (type <value>)
(N)

Returns the data-type of its single argument, return values can be,

    1/VTF_STRING    --
                string
                 type
    2/VTF_NUMBER    --
                number
                 type
```

1.131 unbind

```
                    (unbind {<key>})
(N)      (S)

Remove any
                command string
                 bound to <key>.

See also,
                Command String
                s,
                bind
```

1.132 undo

```
    (undo <type>)
(N)   (S)
```

This commands undoes changes, currently the only type of undo supported is to reset the state of the last-edited line to what it was before being edited, <type> can be,

```
    l --  only undo line if cursor is on the line that the text in the
      undo buffer was copied from.
    L --  always undo line, don't worry if cursor isn't on line to be
      undone.
```

When the undo-buffer is used the contents of the line it's used on is copied into the buffer, what I mean is that undoing something twice gets you back where you started.

This command has a limitation, sometimes after undo'ing a line any bookmarks (or any other remembered coordinates) on that line may be slightly out, this is _not_ a bug just a [mis]feature.

1.133 unsleep

```
                    (unsleep)
(N)
```

Wake up a
                sleep
                ing window.

See also,

                sleep

1.134 while

```
                    (while <condition> <body>)
(N)    (S)         (S)
```

First, <condition> is executed, if it returns a non-zero value <body> is executed and the above steps are repeated, else abort the loop and return the number of times that <body> was executed.

There are a couple of in-built protections against infinite loops, firstly, if the number of iterations reaches a million the loop is aborted. Secondly

```
(and more usefully), the loop can be aborted by sending a ^c break signal to
the editor. This signal can be sent by the break command or, if you have the
software toolkit disks, the breaktask command.

example,
    (while {(move 'dn' 1)}
    {
  ...do something...
    })

See also,

                Command String
                s,
                dowhile
                ,
                if
                ,
                select
```

1.135 Provided Macros

```
                Provided Macros.
****************

Some scripts of macros are included in the distribution (in the macros/
directory, you should copy them to s:jed/). These are mainly intended to
serve as examples of how you can program JEd to meet your own needs.

Macro files,


                blockstack

                stackwins

                make

                indent
                To install any of the sets of macros you have to execute its file  ↩
                   as a JEd
script, use the command,

    (
                script
                 'x' <filename>)
```

1.136 blockstack

```
                blockstack
==========
This file provides commands for a stacking cut/copy & insert, the commands
are,

    (stkcopy <section>)
(N)       (S)

Copy the text in <section> onto the stack.


    (stkcut <section>)
(N)      (S)

Same as stkcut except the copied text is then deleted.


    (stkins)
(N)

Insert the text from the top of the stack.

See also,
    Section Types,
                copy
                ,
                cut
                ,
                insert
```

1.137 stackwins

```
stackwins
=========

This is a command to arrange a specified set of windows into either a
vertical or horizontal stack (ie, adjacent to each other).

    (stackwins <direction> <type>)
()          (S)       (S)

<direction> is either "x" or "y", this specifies which direction to stack
them in. <type> can be,

    a --  do all windows
    v --  do all views of this file
    f --  do one view of each file
```

1.138 make

```
make
====

This is a macro to asynchronously run make (or dmake) in a separate window.
```

```
    (make <args>)
()     (S)

<args> are passed straight to the make utility, by default this macro uses
dmake, though it's very easy to change it.
```

1.139 indent

```
indent
======

This file installs some keybindings to automate the indentation of C code,
it is also suitable for writing JEd scripts with.

If you type the following line (<..> is one keypress, spaces aroud <..> are
just for readability),

    if(expr) <alt {> break; <alt }>

you would get,

    if(expr)
    {
  break;
    }*

The cursor would be left where the asterisk is.

So, you type "alt {" to begin a new block and "alt }" to skip to the end of
the current block.
```

1.140 History

```
                History.
********

Revisions:

                2.07

                2.06b

                2.06

                2.05

                2.04

                2.03

                2.02
                2.01

                2.0
                warning:
    Revision history is updated in realtime -- as soon as a change is
    working I note it in here. This may lead to some disjointed, incorrect
    or just totally weird text.
```

1.141 2.07

```
                2.07 (16-Jan-93)
================

* uses ReadArgs() interface to allow for standard AmigaDOS style command
line parsing. Some more options can be specified from the command line
(pubscreen, tabsize).

* added clause type ^x to allow easy use of control codes.

* added (
                info
                 'barheight') command

* new command (
                saveprefs
                ) to enable preferences to not be saved on exit.

* fixed bug with >= and <= commands being swapped.

* made main document into AmigaGuide format

* added
                renamesym
                 command. Now you can easily add features to primitive
commands.

* fixed a killer bug. before, any attempt to assign a string to a variable
with the = command didn't bother to make a copy of the string, just used the
pointer it was given, deadly stuff, how come I only just found it :-(

* fixed bug in 'ef'
                section type
                , it included one line to many.

* fixed problem of activated view not being refreshed when another view of
the same file closes.

* added
                car
                 and
                cdr
                 commands.

* rewrote a lot of command interpreter. now local variables are much more
```

useful (they can be used anywhere, not just in macros :). Also symbol
clauses which don't have any arguments don't have to be parenthesized, ie,
all variable referencing.

*
addsym
command has been RADICALLY altered, now any type of symbol, local
or global, with any type of value (excluding functions) can be created. This
means you can have such things as local macros :)

* removed global, local and macro primitives, they're now implemented as
macros in the startup file.

* new command, export, for fiddling with the scope of local symbols

* oops, call to AslRequestTags() in cmd_
freq
() didn't mark the end of its
taglist, until the changes detailed above were added the stack must have
been protecting me ;^)

* cleaned up a lot of the return values (now they all conform to what I set
out in the documentation!)

* the commands which can take more than one set of arguments now abort if
any argument is of the wrong type, not just go on to the next set like they
used to.

* new command,
type
, for examining the type of a symbol.

* fixed problem of not being allowed to
insert
sections when you should be
able to (mainly when trying to insert part of a line onto the same line).

* fixed bug in 'mX'
section type
* added
undo
command, only does single-line undo at the moment.

* added
isspace
command

* new option on command line "CD/K"

* added Un*x style regular expression search & substition, uses the
regexp(3) package (with a couple of modifications).

1.142 2.06b

```
                2.06b (10-Dec-92)
=================

* fixed the failure to open initial windows on the screen named by (
                setpref
                `p')
```

1.143 2.06

```
                2.06 (04-Dec-92)
================

* added some more stuff to
                title bar
                 display

* added
                block
                -marking from the mouse, double-click to toggle block marks.

* added preference option to allow you to specify when (if at all) to
snapshot the window's position.

* changed
                ARexx
                 message port naming convention

* added
                substr
                 command

* fixed wordwrap crash when you type further than the right margin

*
                position
                 command no longer
                unsleep
                s
                sleep
                ing windows, just sets it so that
when they wake up they get the new dimensions.

* added support for opening on named public screens
```

1.144 2.05

```
                2.05 (22-Nov-92)
================

* fixed below-mentioned reference problem, new reference type to specify the
_line_ to move to.
                makerefs
```

```
                 now uses this when referencing .h files.

* removed 'oldprefs' preference setting, it's intended use has been built
into the main editor code :-) Preference settings embedded in filenotes now
work much more smoothly, the previous preference settings (last only) are
put into the next window opened or the next file loaded or into the
configuration file.

* added new 'm'
                section type
                 (from cursor to a specified bookmark).

* insert command now sets the automark before moving the cursor.

* added DOS variable support, local and env: variables are looked for if an
internal symbol can't be found (they are checked for just before looking for
an ARexx macro).
```

1.145 2.04

```
                2.04 (03-Nov-92)
================

* new
                section type
                 `n'

* oh dear,
                getref
                 command has problems when a single seek offset is given
and the referenced file contains tabs. Since jed expands all tabs into
spaces when it loads the file the seek offset (which isn't used until the
file has been loaded) is incorrect.

* disktab preference setting is now local to each window

*
                clear
                 command now resets titles of sleeping views like it should

* put in some new preference options, mainly to allow individual files to
keep their preferences in their filenotes (I know filenotes are supposed to
be for the use of users only but I've done it in a nice, user friendly,
manner ;-)

* opening new windows from a
                sleep
                ing window no longer crashes spectacularly
(or at all :-)

* when (savetabs == 2) single spaces are no longer `optimized' into tabs

* revised pubman's command syntax
```

1.146 2.03

```
                2.03 (13-Oct-92)
================

* fixed slight cosmetic bug in
                replace
                 command.

* added
                isalpha
                , etc, commands

* fixed
                makerefs
                 (again) and now it references function definitions which
don't have the /*ref*/ heading that was previously required.

* added code to try and call REXX macros for unknown symbols, means that you
can refer to REXX scripts in the same way as JEd macros and commands (except
you can't return results from REXX)

* made all invocations of JEd have their own
                ARexx
                 port.

* added (
                info
                 'port') command
```

1.147 2.02

```
                2.02 (03-Oct-92)
================

* oops, the
                setmenu
                 command didn't actually look for the END keyword, it
gave an error and aborted if it was given.

* added mousebutton qualifier support.

* fixed some of
                arg
                 command

* sigh... the OpenWindowTags() call in windows.c didn't mark the end of its
tag list with a TAG_END, now it does.

* all windows are now opened on the default public screen.

* included the pubman utility to make it easy to give JEd it's own screen

* new commands,
                ^
                ,
                ^^
                ,
                >>
                ,
                <<
                *
                savesection
                 now saves tabs if savetabs is on.

* savetabs preference improved.

* now configuration data is automatically saved by DICE. As a result of this
the saveprefs command has been discarded.

*
                setmenu
                 didn't recognize SBAR, it looked for SUBBAR... now it's fixed, I
suppose I shouldn't have written the documentation from memory :-(

* when the name of a
                sleep
                ing window changes (
                openfile
                ,
                rename
                , etc...) it's
title is now corrected.

* more of the
                move
                 commands set the auto-mark now.

* added (
                move
                 'of') command.

* IMPORTANT: before, reference seek positions started at 1, now they start
at 0. This means all .jrefs files containing seek offsets have to be remade.
This is to provide consistency with the (
                info
                 'offset') command and (
                move
                'of').
```

1.148 2.01

```
                2.01 (20-Sep-92)
================

* removed (undocumented) 8 argument limit on commands - now commands can get
as many arguments as memory allows. (not quite,
                format
                 and
                req
                 are limited
to 20 formatting values).

* fixed bug in (saveprefs) command, didn't show up until version bump.

*
                position

                unsleep
                's a sleeping window before changing its position.

* fixed
                activatefile
                's refresh problems.

*
                makerefs
                 failure to expand a a pathname when no -full is given has been
fixed. also, structure searching has been rewritten, it won't give 100%
success but will work on the system include files (or at least most of
them). typedef'ed structures are referenced by their typedef not any given
structure tag. makerefs can now be interrupted by ^c

* new commands,
                break
                ,
                select
                ,
                dowhile
                ,
                symboldump
                ,
                move
                 `bs', move `be',

                info
                 `size', info `offset',
                script
                 `s',
                ilock
                , info `asleep',
                setpref
                `scrollhack',
                changes
                * made
                poke
                 and
                replace
                 `r') increment change count

*
                extract
                 can now get any amount of text (the whole file if you want!)

* the libraries; commodities, iffparse and asl are only kept opened when
needed. For commodities and iff this is only when they're being used. asl is
```

```
kept open after the first use of the file req. Under V37 iprefs keeps
iffparse open all the time anyway so this doesn't help much.

* wow, now commodities is kept closed, scrolling is faster. (as long as no
one else has cx open.) (i expect that this is only noticeable if you're
running a 68000 and v37 or less).

* oops, opening and closing the commodities.library so much sometimes
crashed the input.device. Wrote my own description -> key/qual bytes
function. See the (bind) description for more info. commodities isn't used
at all anymore.

* added '
                global
                ' as synonym for '
                addsym
                ', fits better with
                local
                .

* some commands now take more than one set of arguments at once (
                bind
                ,

                addsym
                ,
                addpath
                , etc...)

* fixed bug in (
                move
                 `nw') and (move `pw')

* scrolling speed doubled (when no block is on screen) due to a cunning tip
from Adriaan van den Brand.

* fixed bug when
                delet
                ing from column 0 to column n (NOT 0). Line isn't
joined anymore.

* can now be started via the Workbench.

* fixed bug of \t characters being the size of disktab not tabsize when

                insert
                ing strings or characters.

* fixed some commands who didn't bother to check their arguments before
using them.

* fixed a bug in the menu creation function and added some new menus to the
jed-menus script.

* added code to create backups for each saved file, a configurable number of
backups can be stored in a configurable directory.
```

```
* oops, the command template checker didn't detect errors when a command was
passed an incorrect _number_ of arguments. Now it does.

* added
                null clause
                 (@)

* fixed up readtx() (function to load text), removed possible array over-
stepping and made it so that lines up to the maximum of 32768 chars can be
read properly.

* added one-line history buffer to prompt stuff.
```

1.149 2.0

```
2.0 (08-Sep-92)
===============

* total rewrite from version 1, switched from assembler to C.
```

1.150 Known Bugs

```
Known Bugs.
***********

System requesters (Please insert volume, etc...) only open on the correct
screen while the default public screen is the same as what we're open on.

The makerefs program is pretty inadequate (but slowly getting better :-)

Won't work satisfactorily on a single bitplane screen.
```

1.151 Contact Address

```
Contact Address.
****************

e-mail:
    jsh@ukc.ac.uk
    (hopefully valid until July '95)

paper-mail:
    John Harper,
    91 Springdale Road,
    Broadstone,
    Dorset BH18 9BW,
    ENGLAND.

If you can, please use email, if you don't get any reply it's because its a
University holiday and I'm at home, so try the snail-mail address.
```