NAME

krc – Kent Recursive Calculator

SYNOPSIS

krc [*options*] [*file*] [*arguments*] ...

OPTIONS

-h *cells* Specify the size of the heap. On a 32-bit computer, each cell requires 16 bytes, on a 64-bit computer 32 bytes. The default is 128000 cells.

-d *bytes*
Specify the size of the dictionary, or "string space", used to store identifiers, comments and strings. The default is 64000 bytes.

-n Don't include the prelude of standard definitions.

-l *lib* Use *lib* in place of the prelude.

-L Use a legacy prelude from 1981 in place of the prelude.

-s Load prelude or *lib* stripped of comments (saves string space).

-c Turn on reduction counting (the same as the **/count** command).

-g Turn on garbage collector statistics (the same as the **/gc** command).

-o Turn on the printing of compiled objects (the same as the **/object** command).

-z Sets base for list indexing to 1 instead of 0 (legacy option).

-e *expression[?!]*
Instead of being interactive, evaluate *expression* and display its value (with '?') or flat-print the value (with '!').

If a file name is given after the options, its contents will be loaded as the *current script*. If you don't give a file name the current script is initially empty.

Arguments following the file are permitted only when a **-e** option is present and in this case the file name and any arguments are placed in a list of strings called "**argv**".

DESCRIPTION

Kent Recursive Calculator was one of the earliest higher-order, lazy, purely functional programming systems. Written around 1980, it was a simplified version of its author's language SASL, developed at St Andrews University in 1972-6, and in turn was succeeded by Miranda, the main precursor of Haskell. These days it has the advantages of being a simple language of this kind capable of running in very little memory (75KB code plus whatever space is allocated for heap and dictionary).

A KRC program is called a *script*, which consists of a sequence of *definitions* each of which consists of an optional comment followed by one or more *equations*, one per line.

When you invoke krc, it loads a set of standard definitions called the *prelude*, then prints a welcome message followed by a prompt, **krc>**. You have now entered a KRC *session*. In a session you can type commands, expressions to be evaluated, and definitions to be entered in the current script. A new prompt appears as each task is completed, until you end the session.

COMMANDS

The interpreter accepts a range of commands to inspect, edit, load and save scripts, as well as a few system-oriented things. Each command is entered on a single line, followed by [Enter].

The command to inspect a definition, say of 'foo', is just the name on a line by itself

```
foo
```

the system responds by displaying the definition associated with 'foo', which may be in the prelude or in the current script. If the definition has more than one equation these are shown numbered, from **1**, the numbers are used when editing definitions.

You can display a section of the script, say from the definition of 'foo' to 'bar' by typing foo .. bar

The other commands all start with a / character.

/ Displays all of the current script

/quit Ends a KRC session. Shorthand: **/q**. Typing control-D also has this effect.

Be aware that ending a session does not automatically save the current script, for that you must use **/save** (see below).

In the commands which follow '*name*' means one of the names defined in the current script;

Where it says '*name(s)*' you can put a name, several names separated by spaces, or a range *foo***..***bar* which refers to definitions *foo* through *bar* of the current script, or *foo***..** which means from the definition of *foo* to the end of the script.

Where it says '*part(s)*', this refers to equation numbers from a definition in the script, for deletion or reordering.

A *part* can be a single number like **1**, a range like **2..4**, something like **4..**, which means from equation **4** to the end of the definition; *part(s)* means a list of these separated by spaces.

/delete *name(s)*
Deletes one or more definitions from the script. Shorthand: **/d**
If no *names* are supplied, it deletes the whole script (**beware!**).

/delete *name part(s)*
Deletes the numbered equations from the definition of *name*. To delete the comment, enter an empty comment for the same name, for example "foo:-;"

/reorder *name name(s)*
Place the definitions of one or more *names* immediately after those for the first mentioned *name*, in the order shown.

/reorder *name part(s)*
Reorder the equations within a definition by moving the numbered equations listed in *part(s)* to the top of the definition, in the order shown.

/aborder
Sorts the script's definitions into alphabetical order.

/rename *from***,***to*
Change the name '*from*' into '*to*', in all the places where it is used in the script; *from* and *to* can both be lists of names separated by spaces, in which case the first name in *from* is changed to the first name in *to* and so on. This allows you to swap two or more names in the script without losing any of them.

/save *filename*
Saves the script in the named file. If *filename* is omitted, it saves to the last filename mentioned.

/get *filename*
Adds the contents of a file to the script. If *filename* is omitted, from the last filename mentioned.

/file Shows the current default filename. Shorthand: **/f**

/file *filename*
Changes the default filename.

/list *filename*
Lists the contents of a disk file, like Unix's "cat" or DOS's "type" commands. If the *filename* is omitted, it reads from the last filename mentioned.

/dir List the filenames in the current directory or folder, like Unix's "ls" or DOS's "dir" commands.

/names Displays the names defined in the script.

/lib Displays the names defined in the prelude.

/openlib
Allows you to modify the definitions of the prelude (they are normally write-protected). Any changes are made to the copy of the definitions held in memory during the session - only if you use **/save** (filename) do they get written to a file.

/clear Clear all memo fields. When a simple definition in the script is expanded i.e. one like **foo** = STUFF, the value is stored in a memo field for **foo** so STUFF need not be evaluated again. These are cleared automatically when any definition in the script is changed.

/count Turns on reduction counting.

/gc Enables the printing of memory usage information each time a garbage collection happens.

/reset Turns off reduction counting, garbage collection information and object display (see below).

/dic Report current state of string space.

/help Prints a list of help topics. Shorthand: **/h**

/help *topic*
Displays the relevant helpfile, using the UNIX command '**less**'. This allows you to move backwards (type 'b') in the file as well as forwards (type SPACE, or ENTER to move forward one line at a time), type 'q' to quit. Say '**man 1 less**' on the UNIX command line for more details.

These last commands are for use in debugging the system:

/object Displays the internal representation of each expression before evaluating it.

/lpm The "List Post Mortem" displays the KRC machine's internal state.

EXPRESSION EVALUATION

An *expression* is a mathematical formula denoting a value. For details of what can be in an expression see later sections: IDENTIFIERS, DATA TYPES, OPERATORS and ZF EXPRESSIONS. To evaluate an expression in a KRC session and print the result on the screen, type the expression followed by '**?**'.

For example, you could type **2+2?** to obtain **4**.

If the value is a data structure it is printed as you would enter it, with quote marks around strings, square brackets around lists and commas between list elements. For example ('++' joins lists together):

```
krc> [1..3]++["...\n"]++[2*2]?

[1,2,3,"...\n",4]
```

Note that with '**?**' control characters in a string are shown, not obeyed ("\n" means newline).

An alternative method of printing uses '**!**' in place of '**?**'.

```
krc> [1,2,3,"...\n",4]!

123...
4
```

'**!**' prints strings without quotes, obeying any control characters, numbers are printed in the usual way and with lists '**!**' scans left to right, recursively scanning any sublists encountered, printing only the atoms (numbers and strings). The effect is as if the data were squashed flat and joined up into a single string (although '**!**' does not actually use any additional string space).

As a general guide '?' (print values to show structure) is used when developing and testing functions and '!' (flat-print) for the final output of a program where you want to control layout. Note that '?' adds a trailing newline while '!' doesn't.

The reader may find it helpful to study the function **show** defined in the prelude;
x?
is actually equivalent to **show x : ["\n"]!**

Reminder: For KRC to print the value of an expression it must be followed by '**?**' or '**!**'.

THE SCRIPT: COMMENTS AND EQUATIONS

A KRC script consists of a sequence of *definitions*, each of which is composed of an optional *comment* followed by one or more *equations*. The order of the definitions has no significance as they are all in scope throughout the script, but the order of the equations within a definition may affect its meaning, as they are tried in the order written.

A comment is the name, of a function or other data item, followed by the two characters "**:-**" followed by any number of lines of text and terminated by the first semicolon "**;**".

```
  flub :- here is some text
       ... explaining "flub";
```

An equation is one line of a definition and has this form:

LHS = expression

or

LHS = expression, guard

where **LHS** ("left hand side") is a name optionally followed by formal parameters and **expression** describes the value which the lhs assumes in this case.

The optional **guard** is a Boolean (i.e. truth-valued) expression which determines if the equation is applicable. If the guard evaluates to "TRUE" the equation is used, otherwise it is not.

Note that equations are written one per line (no terminating semicolons needed, or allowed). Examples

```
radix = 10
girls = ["susan","jill","pandora"]
```

are simple definitions. A simple definition has just a name as its LHS.

We can define a function by introducing a formal parameter

```
sq n = n * n
```

The formal parameter 'n' is a local variable of the equation. Any name can be used as a formal parameter, it does not have to be a single letter. However name clashes, like using 'sq' as a formal parameter when it is also a top-level name, are best avoided (occurences of the name on the right of the equation will refer to the formal parameter).

Names which appear at the head of an LHS, which to this point are 'radix', 'girls' and 'sq', are known as *top-level names* and have the whole script as their scope, as do the names defined in the prelude. (The scope of a name is the region of text in which it can be used with the same meaning.)

Here is a function of two parameters

```
sqdiff m n = m*m - n*n
```

We call it by writing e.g.

```
sqdiff 2 3
```

not sqdiff(2,3) as you would in some other programming languages.

More complex function definitions may require several equations, for example

```
fac 0 = 1
fac n = n * fac(n-1)
```

defines a factorial function which can then be used, for example by writing

```
fac 10?
```

Note the use of a constant, 0, as formal parameter in the first equation. This is one way to do case analysis.

Guards can also be used to do case analysis. A safer version of 'fac' would be:

```
fac n = 1, n <= 0
    = n * fac(n-1)
```

Here we use a guard, 'n <= 0', to stop 'fac' going into a loop on negative numbers. Entering an equation with no left hand side means that the LHS is the same as in the line previously entered.

We have seen that a formal parameter can be a name or a literal constant. It can also be a pattern to match list structure as in this example – 'rotor' carries out a simple rotation on lists of length 3.

```
rotor [a,b,c] = [b,c,a]
```

To match lists of unknown length we use a pattern involving ':', for example the functions 'hd', 'tl', which return the first element of a list, and the remainder of the list without the first element, are defined

```
hd (a:x) = a
tl (a:x) = x
```

Another example is this function which copies lists of length <=2 and extracts the first 2 elements of lists with length >2.

```
take2 [] = []
take2 [a] = [a]
take2 (a:b:x) = [a,b]
```

Note that ':' is right associative so 'a:b:x' means 'a:(b:x)'.

In summary, an LHS consists of the name being defined followed by zero or more formal parameters. Each formal parameter can be

a *name* like **x** or **height**
a *literal* number or string like **3** or **"bill"**
a *pattern* like **[p,q,r]** or **(a : x)**
where any of the p, q, r, a, x could themselves be a name, literal or pattern.

Names introduced by formal parameters are local to the equation in which they appear.

An equation with one or more patterns in its LHS applies when the actual parameters of a function invocation match the corresponding formal parameters; that is they have the same list structure and, where the pattern contains a literal, the values are equal. If there is also a guard, that must evaluate to "TRUE".

Where a definition has more than one equation they must all have the same number of formal parameters. Notwithstanding this, it is possible in KRC to define functions which take a variable number of arguments, because the right hand sides can be of varying type. (Terminology – the actual parameters of a function are also called its *arguments*.)

KRC supports the interactive development of scripts.

To enter definitions in the script during a session you simply type in the equations and comments at the prompt, one at a time. You can then enter expressions to test your definitions and edit them (i.e. delete, reorder etc) using the COMMANDS listed earlier and re-enter equations as needed. The line editing facilities provided by "linenoise" and/or the use of a mouse to cut and paste make this task relatively easy.

If you enter an equation with the same LHS and guard as an existing one it will replace it. Entering a comment will replace an earlier comment on the same name. To delete a comment enter an empty replacement

```
name :-;
```

Another way of proceeding is to use an editor to create the script and then start a krc session with it as the file. The KRC system was designed to allow it to be used without a separate editor (back in 1980 screen editors like **vi** and **emacs** were not widely available as they are now).

IDENTIFIERS

The names used in scripts and expressions are built from **A-Z a-z 0-9** underscore (_) and apostrophe (´), start with a letter and can be of any length. Case is significant, so X and x are different identifiers. Note that KRC has no reserved words.

DATA TYPES

KRC has four data types: numbers, strings, lists and functions.

The four types of value have the same "civil rights" – any value can be (i) named, (ii) included in a list, (iii) passed to a function as its argument, (iv) returned by a function as result.

There is no static type system – values of different types can be freely mixed, for example as elements of a list and a function can accept arguments of more than one type and return results of different types for different inputs. Type errors, e.g. trying to multiply two strings, are detected at run time. Languages of this kind (LISP is another example) are often called 'typeless', but *dynamically typed* is more accurate.

A value can be tested by the four functions 'number', 'string', 'list' and 'function' and returns "TRUE" for one of these, "FALSE" for the others. There is also a function 'char' which recognises strings of length 1 and a function 'bool' which reconises the strings "TRUE" and "FALSE".

Numbers are integers (whole numbers, positive, negative and zero) written in the usual decimal notation, e.g. 3 -17 500

Strings are sequences of characters enclosed in double quotes, e.g.

    "cattle" "oh my!"

Control characters are written using '\', e.g. "\n" is newline. The escape conventions are as in C

    \a (bell) \b (backspace) \f (formfeed) \n (newline) \r (carriage return) \t (tab) \v (vertical tab) \\ (backslash) \' \"

and numeric character codes

    \ddd (up to three decimal digits).

KRC character codes are in the range 0–255 (of which 0–127 are ascii, the meaning of codes above 127 depends on the *locale*).

Note the strings "TRUE", "FALSE" are used as truthvalues, there is no separate Boolean type.

KRC has no separate type 'character' distinct from 'string'. A character is just a string of length 1, e.g. "a" (you cannot write this as ´a´, that would not be recognised).

The prelude function 'explode' converts a string into a list of characters, e.g.

```
krc> explode "hello"?
["h","e","l","l","o"]
```

KRC treats strings as atomic; they are stored uniquely in packed form. Strings can be tested for equality or order without unpacking them but to access the internal structure you must use 'explode'.

There is a function 'implode' which compresses a list of strings into a single string, but this is less often needed. Processing of textual data is typically done on lists of characters (or sometimes trees) and given '!' there is no need to convert the result back into a single string before printing.

Lists are written using square brackets and commas between elements, e.g. [1,2,3,"eric"]. Note the empty list, [] and singletons, e.g. [3].

An important operation on lists is '**:**' which prefixes an element to the front of a list. So 'a:[]' is '[a]', 'a:b:[]' is '[a,b]' and so on. Note also the '**++**' operator which appends one list to another. The elements of 'x++y' are those of 'x' followed by those of 'y'.

As some, or all, of the elements of a list can be lists it is possible to construct multi-dimensional arrays (as lists of lists of ...) and more generally trees, of any shape. Lists can be of infinite length, for example

```
ones = 1 : ones
```

is an infinite list of ones and

```
abyss = ["down",abyss,"up"]
```

is a tree of infinite depth. These examples are possible because of KRC's general strategy of *lazy evaluation*, which includes not evaluating the parts of a structure until they are accessed.

There is a notation '[a..b]' for a sequence of consecutive integers, e.g. '[-10..10]' is a list with 21 elements. This notation also has an infinite form, e.g. '[2..]' is a list of the integers starting at 2.

Similarly the notation '[a,b..c]' can be used for an arithmetic sequence running from a to c with step (b-a). E.g. [1,3..99] are the odd numbers from 1 to 99. Again this has an infinite form, e.g. [0,-5..]

A list can be applied to a non-negative number, n say, to extract its n'th element (counting from 0). So given

```
x = ["mon","tue","wed","thu","fri"]
```

then 'x 0' is "mon", 'x 6' is "fri", and 'x 7' is an error. A list can therefore be supplied in a context where a function is expected, for example as an operand of '.' the functional composition operator.

Functions: a function calculates an output value from given input value(s). The rules of calculation are given by its defining equations (except for a few built-in functions defined in machine code).

Applying a function to its input values (or *arguments*) is done by juxtaposition, e.g.

```
f x y
```

If an argument is an operator expression or another function application it needs parentheses to force the correct parse. E.g.

```
f (g x) (y+2)
```

A prefix or infix operator in single quotes denotes the corresponding one- or two-argument function, e.g.

```
ops = [´+´, ´-´, ´*´, ´/´]
```

is a list of the four functions of arithmetic. See below, under OPERATORS.

Continued application: a function application can return a function, e.g. (ops n) is a function, which one depends on 'n'. We can immediately apply this to its arguments, thus

```
ops n x y
```

We could write '(ops n) x y' but the parentheses are not needed as function application is *left associative*.

Partial application: a function of two or more arguments can be applied to a smaller number of arguments and the result is a function that is waiting for the remaining arguments. For example

```
double = ´*´ 2
```

is a function that can be applied to a number and will multiply it by 2.

Partial application works because KRC uses a model of functions derived from *combinatory logic*. In this model, all functions actually take a single argument. An n-ary function, for n>1, is a function that, when applied to its argument, returns an (n-1)ary function that is waiting for the next argument. So taking the case n=3 as an example

f a b c

means

((f a) b) c

but because of the left-associative rule the parentheses are normally omitted.

OPERATORS

The following table lists KRC's prefix and infix operators in order of increasing binding power.

The second column gives associativity:- a left associative operator, e.g. '-' has

a - b - c ==> (a - b) - c

while a right associative operator, e.g. '++' has

a ++ b ++ c ==> a ++ (b ++ c)

Infix operators in the same row have the same binding power and are jointly right or left associative.

The third column gives a brief statement of meaning. Where needed the notes below add further detail.

operator	**associativity**	**meaning**
: ++ --	right	list prefix, append, listdiff
|	right	logical *or*
&	right	logical *and*
\	prefix	logical *not*
> >= \= == <= <	see below	comparisons
+ -	left	add, subtract
+ -	prefix	plus, minus
* / %	left	multiply, integer divide, remainder
** .	right	exponent, function-compose
#	prefix	length of list

The application of a function to its arguments binds more tightly than any operator. Parentheses **(...)** can be inserted freely in expressions to enforce the intended parse or to make it clearer to the reader – redundant parentheses do not change the meaning (this is true both in expressions and in formal parameters).

The operators **:** (prefix) and **++** (append) have been discussed earlier, in the section on DATA TYPES. The expression **x -- y** returns a list of the elements of **x** without those that appear in **y**, the exact definition is given by the prelude function **listdiff**.

The logical operators **|**, **&** do not evaluate their second operand if this is not needed.

a | b

returns "TRUE" if **a** is "TRUE", otherwise it returns **b**. Similarly

a & b

returns "FALSE" if **a** is "FALSE", otherwise it returns **b**.

The relational operators **>**, **>=**, **==**, **<=**, **<** have a special syntax which allows continued relations, as commonly used in mathematics books. For example

> **0 <= x <= 10** means **0 <= x & x <= 10**
> **a == b == c < 0** means **a == b & b == c & c < 0**

The operators **==** and **\=** can be used to test for equality or inequality on any pair of values, returning "TRUE" or "FALSE". Values of different types are always unequal. Two lists count as equal if and only if they are of the same length and (recursively for sublists) corresponding elements are equal.

Equality tests between functions give unpredictable results and should not be used. (Example: '**map == map**' has value "TRUE" but '**map abs == map abs**' is "FALSE".)

The order testing operators **> >= <= <** work on numbers in the usual way and on strings compare according to the C function *strcmp()*. For words all in upper case, or all in lower case, this amounts to dictionary order, but be aware that upper case letters precede lower case letters, so "ZEBRA" < "ant". It is an error to test lists or functions for order, or two values of different type.

+ - * / are the usual arithmetic operations, with **%** for remainder. KRC arithmetic deals only with integers thus '/' discards the fractional part, rounding towards **0**, while **a % b** takes the remainder on dividing '**a**' by '**b**', the sign of the result being that of '**a**' (these rules for integer division and remainder are the same as C).

Prefix '**-**' reverses the sign of its integer operand (leaving **0** unchanged). A negative constant like '**-1**' is, syntactically, an operator expression and must be parenthesised when passed to a function, thus '**f(-1)**' whereas '**f -1**' would attempt to subtract **1** from **f**. Prefix '**+**' does nothing and is discarded.

a ** b raises '**a**' to the power of '**b**' which must be non-negative.

The composition of two functions '**f . g**' is defined as follows

> **(f . g) x = f (g x)**

The operator **#** gives the length of a list, for example '**#[1,2,3]**' is **3**. To find the length of a string use the prelude function **printwidth**.

A prefix or infix operator in *single quotes* denotes the corresponding one- or two-argument function. For example **'+'** is the addition function and **'#'** and **'\'** are functions which take the length of a list and the negation of a truthvalue. Note that **'-'** is subtraction, for unary minus you can use the prelude function **neg**.

ZF EXPRESSIONS

ZF expressions are a notation to generate lists from other lists. (These would now called *list comprehensions*, a term coined by Phil Wadler in 1985.)

We start with some simple examples

> **{n*n; n<-[1..]}**

is a list of all square numbers [1,4,9,16,25...]. The expression before ';' is the *body* of the ZF expression while 'n<-[1..]' is called a *generator* and 'n' is a local variable of the ZF expression. The effect is to create a list of the values taken by the body for each value of 'n' drawn from the list given in its generator.

The next example is a function for finding the factors of a number

> **factors n = {r; r<-[1..n/2]; n%r==0}**

This makes a list of the numbers between 1 and n/2 which divide n exactly. The expression after the second semicolon is called a *filter* – only those values of the generator are used for which the filter is "TRUE".

A ZF expression can have two or more generators, so

> **{a+b; a<-[1,2,3]; b<-[10,20,30]}**

gives you a list of nine two-digit numbers.

Our last example finds all positions on a chess board which are a knight's move from [x,y]

> **knights_move [x,y] = {[i,j]; i,j<-[1..8]; (i-x)**2 + (j-y)**2 == 5}**

Searching the whole board in this way is not very efficient, but never mind. Note that 'i,j<-[1..8]' is shorthand for two generators with i, j both drawn from [1..8].

The general form of a ZF expression is

> **{expression; qualifier; ... ; qualifier}**

with one or more qualifiers, each of which is either a *generator*, of the form

```
name-list <- expression
```

or a *filter*, which is an expression whose value is "TRUE" or "FALSE".

If the first qualifier is a generator, which is usually the case, the initial '**;**' can be written '**|**', which aids readability.

For examples of programming with ZF expressions see David Turner's papers "The Semantic Elegance of Applicative Languages'' (1981) and "Recursion Equations as a Programming Language" (1982).

(**Historical note**: The semantics of ZF expressions in KRC differ from those of modern list comprehensions in *interleaving* outputs of multiple generators to ensure that all combinations are eventually reached, even with two or more infinite generators. The motivation was to make them behave more like set comprehensions, whence the use of {}. List comprehensions with square brackets and the now familiar "nested loop" semantics first appeared in Miranda, in 1984.)

THE PRELUDE

When KRC is started it first loads the **prelude**, a library of standard definitions, most of which are in KRC but some are in machine code, including functions to access the operating system (e.g. **read**, **write**).

To list the names defined in the currently loaded prelude during a KRC session, type

/lib

You can inspect any of the definitions by typing the name, e.g.

map

Each has an comment explaining what it does, in addition to its defining equations.

The standard prelude is usually at **/usr/lib/krc/prelude** but may be placed elsewhere by the installer. To view the standard prelude in a KRC session, say "**/help prelude**".

Another library in place of the standard prelude can be specified on the command line when KRC is invoked by the option **-l** *lib*. If you invoke KRC with option **-n**, no prelude is loaded.

If KRC is invoked with option **-L**, a legacy prelude is used in place of the standard one. To view this in a KRC session, say "**/help lib1981**".

Note that without a prelude you cannot use ZF expressions, whose implementation requires **interleave** nor can you use the '**--**' operator, which requires the function **listdiff**.

LINE EDITING AND ESCAPE TO UNIX SHELL

Line editing: if KRC is compiled with the "linenoise" module, the arrow keys and other keys can be used to scroll back and forth through lines typed in a KRC session, to edit and resubmit them. For a summary of available key bindings, with and without "linenoise", say **/help keys** in a KRC session.

Escape to Unix shell: type Ctrl-Z to suspend KRC and return to Unix command line, say **fg** to resume the KRC session.

MAKING UNIX COMMANDS

You can write a Unix command in KRC by making its script executable through the "magic string" mechanism. To do this, add a line as the start of the script, of the form:

#! /usr/bin/krc -e *expression*!

where *expression* is to be evaluated when the program is run. This can be a KRC expression of any complexity, terminated by '**?**' if its value is to be shown with its structure or '**!**' if it is to be flat-printed.

Other flags such as "**–l** *lib*" for an alternative prelude, "**–h** *size*" for a larger heap, can precede "**-e**".

The name by which the script is called, followed by the command-line arguments that the user supplies when it is run, are placed in a list of strings called "**argv**".

Example: A simple command to display the contents of files. Put the following 2 lines in a file called "list".

```
#! /usr/bin/krc -e main!
main = map read (tl argv)
```

then make it executable

```
chmod +x list
```

now you can use it, for example

```
./list list
```

will display the contents of the file itself.

Any KRC script can be made executable by placing a "magic string" at the start. When used as the script in an interactive KRC session an initial line in '#!' is treated as a comment and ignored; it has effect only if the file is invoked as a command on the Unix command line.

LANGUAGE CHANGES

The KRC language described here is as David Turner designed it around 1980, except for the following changes.

The base for indexing lists has been changed from **1** to **0**. The option **-z** reverts to lists being indexed from 1 (this may be needed for legacy KRC scripts dating from the 1980's).

A revised prelude has been provided, with some more efficient definitions (e.g. **reverse**, **sort**) and a more modern collection of functions. A legacy prelude from 1981 is also available (option **-L**).

Escape characters in strings, "\n" etc., as in C, have been added. KRC as originally designed used instead names for special characters: **nl np quote tab vt**, these remain in the prelude. Example: instead of **"123\tgo\n"** you can say **["123",tab,"go",nl]** to get the same effect with "!".

A distinction has been introduced between '**=**', definitional equals and '**==**' the equality test (KRC originally used '**=**' for both).

In-line comments, from "||" to end of line, have been added, thus

```
|| this is a comment
```

these don't fit very naturally with KRC's model of interactive script development as they are discarded when entered during a session. But the facility is useful for scripts prepared with an editor as it allows comments on individual equations and on the whole script.

LEGACY SYNTAX

With option **-z** these alternative operator forms are accepted and converted to their standard variants:

˜ as a synonym for **** (prefix *not*)
˜= as a synonym for **\=** (*not-equals*)
= as a synonym for **==** (*equality test*)

LIMITATIONS

Guards do not work correctly with simple definitions, for example

```
feb = 29, leap_year
   = 28
```

fails with strange error when leap_year="FALSE". The language as originally defined did not allow guards with simple definitions, only with functions. The syntactic restriction has been removed but the corresponding semantics still needs fixing.

The length of a string limited to 255 characters. The space used by strings created during a session is never freed, mitigated only by the fact that multiple copies of the same string are shared. This means that programs which create a lot of temporary strings as part of their operation will soon run out of the space in which strings, identifiers, file names and comments are stored. Note, however, that KRC's "**!**" output mode removes much of the need to create temporary strings.

Numbers are limited in range to that of the underlying processor's machine word, i.e. they are signed 32-bit or 64-bit integers. Calculations which exceed this range cause a run-time error.

If linenoise is enabled, multiline comments cannot be entered interactively in a KRC session.

FILES

/usr/lib/krc/prelude
The standard prelude.

/usr/lib/krc/lib1981
A legacy prelude dating from 1981 (invoked by option -L).

KRC looks for the prelude first in **./krclib** if present, then in **/usr/lib/krc** (this may be changed by the installer).

SEE ALSO

http://krc-lang.org/
The KRC home page.

Recursion Equations as a Programming Language, D. A. Turner, January 1982
This includes an overview of the KRC language and can be found at the above website.

http://github.com/antirez/linenoise
The "linenoise" line-editing module (the version included with the KRC sources has been lightly edited to configure it appropriately).

AUTHOR

The KRC system was designed by David Turner who implemented it in BCPL for the EMAS operating system from November 1979 to October 1981.

This implementation was created by translating the original BCPL sources into C for the Unix family of operating systems.