Research XFCN: A Free HyperCard Utility

By Ari Halberstadt

ABSTRACT

An external function implementation of a general purpose string searching utility for HyperCard (on the Macintosh). It is possible to search for regular expressions similar to those used by UN*X's egrep, or for plain text using a very fast algorithm. Searches can be done on a single input string, or a list of containers in a single card or in many cards. Output format is flexible, and may be in a special chunk form for further processing by HyperTalk scripts.

Source code in C is included. The program is free; for distribution terms see the appropriate sections in the file “Common Manual”.

This manual assumes some knowledge of scripting in HyperCard.

Table of Contents

Sections

Function descriptions

Functions

Figures

Tables

Scripts

Preface

The choice of a name for a new program is always a delicate matter. On the one hand, one wants a concise and mnemonic name. On the other hand, one does not want to have a name which will conflict with other existing programs. The name “Search” certainly fits the first criteria, yet I'm afraid it fails on the second. There are probably numerous XFCNs out there, all called “Search”, and each one doing something entirely different. Other names which I considered were “grep”, and “egrep”, but the program isn't really the same as either of these programs. The name “find” is already used by HyperCard. I settled on “Research” (short for “Regular Expression Search”), which is certainly mnemonic, not too long, and fairly unlikely to conflict with other names.

Using Research

Calling research

When Research is called with only one parameter it does not search for any text; rather, it interprets the first word of the parameter as a special function to be executed. When Research is called with more than one parameter, it interprets the first parameter as a pattern to search for, the second parameter as the input text, and the third parameter as a list of options which modify the behavior of Research. When called in this way, Research returns (by default) the text that matched the pattern. Thus, Research has two basic forms:

`research(`*function*`)`

and,

`research(`*pattern*`, `*string*`, `*options*`)`

When specifying an option or a function to Research, capitalization of the option or function name does not matter: "FOOD" is the same as "food".

The simplest way to search for a string is:

`get research(`*regular_expression*`, `*input_string*`)`

The *regular_expression* parameter is the expression being searched for. The *input_string* parameter contains the string being searched in. Research returns, by default, the lines that contained text that matched the regular expression. If nothing is found, then nothing is returned.

To specify a list of options, you would use the third parameter. Each option is given as a single word. For instance, to invoke Research using the invert and IgnoreCase options, you would use the following command:

`get research(`*regular_expression*`, `*input_string*`, "invert ignoreCase")`

If you just want to get a string describing the version of Research, you could use only the first parameter, as follows:

```
get research("!")
```

Checking for errors

An error could occur at any time during execution. For instance, Research could have been passed an illegal parameter, or it could run out of memory, or it could be unable to find a container to be searched. If an error occurs, Research stops searching and returns whatever it had found up to the time of the error.

After invoking Research, you should always use the Error function to find out if Research was successful. The Error function will return empty if it was successful, otherwise it will return an error code. You can translate the error code into a descriptive string using the ErrorString XFCN. For instance, the following script checks for an error and halts all scripts if one occurred:

Script 1. **ResearchError**

```
on researchError
      if (research(error) <> empty) then
            answer "Research error:" && ErrorString(research(error))
            exit to HyperCard
      end if
end researchError
```

Monitoring and canceling execution

If Research is executing for longer than about half a second, then it starts displaying a rotating busy cursor, similar to HyperCard's busy cursor. This cursor may be deactivated by specifying the noBusyCursor option.

Execution may be canceled at any time by pressing Command-period. Research will stop searching and will return whatever text it found. The Error function will return the error code ERR_BASIC_CANCELED.

Using Research instead of find

Research can be used in a manner similar to HyperTalk's find command. The script researchNext, which is included in the demo stack, jumps to the next occurrence of a regular expression in a list of containers «script not yet written». This is similar to a command which uses HyperTalk's find command:

```
find chars "hello" in field 1
```

which finds the next occurrence of the string "hello" in field 1. Research is, however, significantly more versatile and powerful than the find command. For instance, Research may search through text which is not inside a field, including any number of named containers, of any type, even scripts of cards and buttons, as opposed to the find command's single background field. In addition, Research supports regular expressions, and offers many options which modify its behavior and the format of the output. Finally, the source code for Research is supplied free of charge; so, if you know how to program in C, you can improve the program and tailor it to your needs.

Regular expressions

Research interprets the pattern string as a regular expression. A regular expression uses a very powerful and concise language for specifying a wide range of character strings. Many characters in a regular expression have special meanings, and so you must be sure you understand exactly what each character does. Characters with special meanings in a regular expression are called “metacharacters”. For instance, the metacharacter period (.) can match any other character, which is more general than the letter A, which can only match another A.

Note: The nometa option turns off the use of metacharacters. Using this option also results in significantly faster searches. All other features of Research are unaffected, so you can skip this section if you just want to search for a string, and don't want to learn about regular expressions.

The regular expressions which Research understands were inspired by the UN*X utilities grep and egrep (grep stands for “Global Regular Expression Printer”, and egrep is the same with “Extended” added). The following descriptions always refer to units of text as the default unit of text, which is a line. By using the separator option it is possible to change the default unit of text.

In the following descriptions of regular expressions, the complete form of a call to Research has been omitted. When a description shows an expression, it means that if Research were given this expression, it would match certain strings. For instance,

```
get research(regular_expression, field "input")
```

is how you would invoke Research to search for a regular expression in the field named "input".

Simple expressions

A simple expression contains no metacharacters, and therefore will match only strings which are identical to the expression. For instance, the following expression finds all places where global variables are declared in a script:

```
"global"
```

Actually, this expression really finds all occurrences of the string "global", so it would find lines such as

```
global variable
```

as well as

```
put "global" after card field 3
```

Matching beginning and end of line

The circumflex **^** and dollar sign **$** metacharacters anchor the pattern to the beginning (^) or end ($) of a line. For example,

```
"on"
```

matches all lines containing the string "on", while

```
"^on "
```

matches all lines starting with the string "on ", and which will therefore probably be script handlers. Finally,

```
"^The only little boy in New York$"
```

matches all lines containing only the string "The only little boy in New York".

The ^ and $ are only treated as metacharacters if they come at the beginning and end, respectively, of a regular expression.

Hiding special meanings

The backslash **** is a special metacharacter which, when it precedes another character, takes away any special meaning the character would have. For instance, if you want a circumflex to appear at the beginning of your pattern, precede it with the backslash, to get "\^". For instance,

```
"\^on"
```

matches all lines containing the sequence "^on". A backslash is also known as the escape character, so preceding a character with a backslash means it's "escaped". Notice that a backslash preceding an open or close parenthesis specifies a tag, while a backslash preceding a digit specifies a tag reference. (These terms are explained below.)

Matching any character

The period is a metacharacter which matches any character at all. For instance,

```
".at"
```

matches the strings "cat" and "bat", as well as "#at" and "!at".

Character classes

A construct called a character class is used to specify a list of possible characters. A character class is introduced with the open square bracket **[**, which is then followed by a list of characters, and finally terminated with the close square bracket **]**. For instance,

```
"[abc]"
```

will match either a, or b, or c. It is also possible to specify a range of characters to match. A range is specified by placing a hyphen **-** between two characters in the class. For instance,

```
"[a-z]"
```

matches the letters a through z, and

```
"[a-zA-Z]"
```

matches all lower case and all upper case letters. The characters which will match a character range are those whose ASCII values fall between the ASCII values of the lower and upper bounds of the range. Thus, the range

```
"[A-z]"
```

matches all upper and lower case letters, as well as all the other characters that fall in that range of ASCII character values:

```
[ \ ] ^ _ '
```

Ranges can be confusing:

```
"[3-68]"
```

matches the characters 3, 4, 5, 6, or 8, not the numbers 3 through 68.

To include a hyphen in a character class you can either escape it (i.e., precede it with a backslash), or place it at the beginning or end of the character class, so that it isn't confused with a range specification. The hyphen is only special within a character class, not outside of a character class.

Finally, if a circumflex **^** immediately follows the opening square bracket of a character class then the character class matches any characters except those in the class. For instance, while

```
"[0-9]"
```

matches any digit,

```
"[^0-9]"
```

matches any non-digit. The circumflex loses its special meaning if used anywhere except the first character of the character class.

Closures – repeated pattern matches

Note: The curly-braces form of a closure may cease to exist in a future version, which will, however, employ a significantly faster regular expression searching algorithm. The "*", "+", and "?" closures will still be available.

A number enclosed in curly-braces **{ }** following an expression specifies the number of times that the preceding expression is to be repeated. For instance,

```
"a{4}"
```

matches 4 occurrences of the letter a. Similarly,

```
"[a-zA-Z]{5}"
```

matches all words with at least 5 letters. Normally, closures apply only to the previous character, so "xy{3}" matches an x followed by three y's, not the sequence "xyxyxy".

The general format of a closure is {n,m}, where **n** is the minimum number of repetitions and **m** is the maximum number of repetitions. A missing **n** is assumed to be one, and a missing **m** is assumed to be infinite. There are a few shorthand metacharacters for expressing common closures, shown in the following table.

Table 1. **Closures**

Metacharacter	Equivalent	Description
*	{0,}	Preceding pattern is repeated zero or more times; this is by far the most common closure.
+	{1,}	Preceding pattern is repeated one or more times.
?	{0,1}	Preceding pattern is repeated zero or once only.

Marking expressions with tags

Tags are used for marking a portion of a regular expression for later reference. Tags are specified by preceding them with the two character sequence **\(** and ending them with the two character sequence **\)**. Each tag is numbered according to the order in which it is encountered in the regular expression, from left to right, and a maximum of 9 tags are possible. A tag is referred to using a tag reference, which consists of a backslash followed by a single digit from 1 through 9. For instance,

```
"A \(simple\) example which is very \1"
```

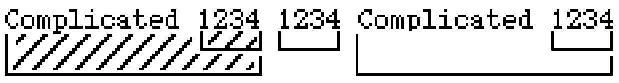
matches the string "A simple example which is very simple". A more useful expression would be:

```
"\([a-zA-Z]+\) +\1"
```

which matches doubled words (a common typing mistake), such as "the the" and "Boom boom". To illustrate how more than one tag is used, consider the following:

```
"\([a-zA-Z]+ \([0-9]+\)\) +\2 \1"
```

which matches a word, followed by a number, followed by spaces, followed by the same number which is then followed by the word and number. The string "Complicated 1234 1234 Complicated 1234" would match this expression. The following illustration shows the range of characters matched by each tag:



The following regular expression searches for palindromes (words or phrases spelled the same way backwards as forwards) with four letters:

```
"\([a-zA-Z]\) *\([a-zA-Z]\) *\2 *\1"
```

This will match "aaaa", "abba", "noon", "no on", "n o o n", etc.

Grouping patterns with parenthesis

Sequences of expressions may be grouped into one large expression by surrounding them with parenthesis **(** and **)**. This is most useful when applying a closure to a large pattern. For instance,

```
"(xy)*"
```

will match a sequence like "xyxyxy", as opposed to

```
"xy*"
```

which matches a sequence like "xyyy". Parenthesis may be nested, so

```
"(Broken (record )+)+"
```

matches sequences like "Broken record " and "Broken record record record Broken record record record ".

Alternate expressions with the vertical bar

Alternative expressions are specified by separating them with the vertical bar **|**. For instance,

```
"This program is (excellent|complicated)"
```

matches the string "This program is excellent" or the string "This program is complicated". The following expression will find all the handlers and functions in a script:

```
"^on |^function "
```

Actually, this expression could be more concisely written as

```
"^(on|function) "
```

Options

If there are three parameters to Research, then the third parameter is interpreted as a list of options. Each option is separated from the other options by spaces. Following is a complete, alphabetical, list of the options.

Chunk

Instead of outputing the complete matched input line, the output is in a form giving the start and end characters of the matched portion of the input line. This form is readily converted into HyperTalk chunk expressions, which have the form "char a to b of line c". However, to conserve space, the connecting words are omitted, so the output will be "a b c". Chunk expressions are output one per line, where the chunk expressions on each line refer to a single match on a single input line. Typical chunk expressions are

```
"1 27 3"

"4 4 5"

"56 56 103"
```

which are equivalent, respectively, to "char 1 to 27 of line 3", "char 4 of line 5", and "char 56 of line 103".

Notice that Research normally finds only the first occurrence of the search pattern in each input line. To find all occurrences, you can use the global option.

Chunks and tags

If there are tagged expressions in the regular expression being searched for, then the first item in each chunk description gives the entire range of text that was matched, and subsequent items give character ranges in the form "char 3 to 15" for each tag. For instance, the following chunk expression

```
"7 54 13,10 15,,,20 30,,,,,"
```

indicates that the regular expression matched characters 7 to 54 of line 13, that tag number 1 matched characters 10 to 15 of the same line, and that tag number 4 matched characters 20 to 30 of the same line.

Chunks and the Separator option

If a separator which is not the return character is used (see description of the separator option) then the units of text are no longer lines. Therefore, chunk expressions will be slightly different. The character positions will be specified as offsets from the start of the input, not the start of the current line. For instance,

```
"123 154 23"
```

means that the pattern matched characters 123 to 154 of the input, and that this was the 23rd input unit (as opposed to input line). Another example shows how this would modify output for an expression with tags:

```
"123 154 23,130 135,,,,140 142,,,,"
```

The first part of this example is the same as the previous example; the second part indicates that the first and fifth tags matched, respectively, characters 130 through 135 and characters 140 through 142.

Chunks and the Containers option

The containers option also modifies the format of a chunk expression. The first item of the chunk expression now has more numbers appended to it (so that there are 4 or 5 words, instead of the usual 3 words). The 4th word specifies the container in which the match was found, and the 5th word specifies the card in which the match was found. The 5th word will only be present when at least one of the startCard, firstCard, or lastCard options are specified along with the containers option. For instance, the following command searches for the pattern "^(on|function)" in the containers "script of button 1", "script of button 2", and "script of field 1" in cards 1 to 4:

```
research("^(on|function)", "script of btn 1,script of btn 2, script
of fld 1", "containers startCard 1 firstCard 1 lastCard 4 chunk")
```

The chunk expressions returned by this command could be:

Table 3. **Example chunk expressions**

Expression	You should interpret as
`1 2 1 1 1`	char 1 to 2 of line 1 of script of btn 1 of cd 1
`1 2 1 2 1`	char 1 to 2 of line 1 of script of btn 1 of cd 1
`1 8 2 3 2`	char 1 to 8 of line 2 of script of btn 2 of cd 2
`1 2 10 3 4`	char 1 to 2 of line 10 of script of fld 1 of cd 4

Containers

The input string is interpreted as a comma separated list of containers whose contents must be searched. For instance, the following command searches the HyperCard containers "field 1 of card 4" and "script of button 1" for the string "wonderful":

```
research("wonderful", "field 1 of card 4,script of button 1",
"containers")
```

The above command is equivalent to — but more efficient than — the following commands:

```
put research("wonderful", field 1 of card 4, "containers") into tmp1
put research("wonderful", script of button 1, "containers") into
tmp2
put tmp1 & tmp2 into result
```

Containers and output format

If more than one container is being searched for, as in the above example, then each output string is preceded by the name of the container (the container's name is terminated with a comma). For instance, if the line "-- this is a wonderful planet" were matched in the container "script of button 1", then the output would be

```
script of button 1,-- this is a wonderful planet
```

If the count option were specified in addition to the containers option, and if 4 matches were found in the container "field 1 of card 4" and 5 matches were found in the container "script of button 1", then the output would be

```
field 1 of card 4,4
script of button 1,5
```

When used with the number option the line number comes after the name of the container. So, if the above example matched line 4 of a container, the output would be:

```
"script of button 1,4,this is a wonderful planet"
```

The StartCharacter option

This option may be used with or without the containers option. It is described here due to its similarity and relevance to the startContainer and startCard options.

The next word following the startCharacter option specifies the first character from which to start searching. The word should be an integer, giving a character offset into the container.

How the search proceeds

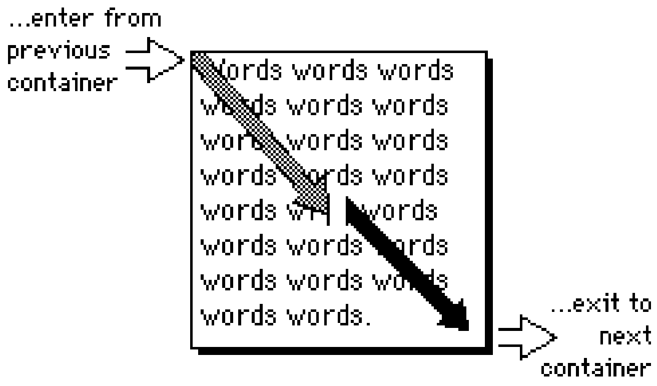
The search proceeds from the start character, up to the end of the input, and then wraps around to the start of the input and continues until the start character is reached. For instance,

```
research(findMe, script of card 1, "startCharacter 500")
```

First skips to character 500, searches up to the end of the script, and then wraps around and searches from character 1 up to character 500.

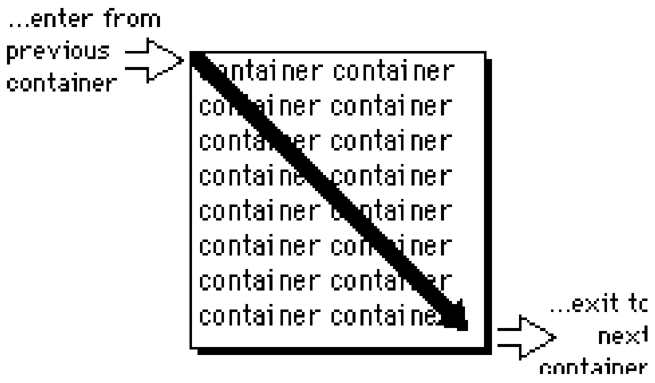
The following illustration should help clarify what is going on. We begin searching at the gap between the gray and black arrows. We then proceed along the path of the black arrow. When the end of the container is reached, we wrap around to the start of the container, and follow the path of the gray arrow. We finally stop searching when we've again reached the gap between the arrows. The “enter from previous container” and “exit to next container” arrows represent what happens when searching more than one container, a process which is elaborated on in the following sections. Contrast this illustration with the next illustration.

Figure 1. **Searching with the startCharacter option**



The second illustration shows what normally happens in a container or input string when the startCharacter option hasn't been used. In this case, the start character is simply zero, so that the gray arrow is non-existent. We therefore only follow the path of the black arrow. Incidentally, this is what happens when we're searching any container other than the container specified with the startContainer option.

Figure 2. **Searching without the startCharacter option**



The StartContainer option

The startContainer option is very similar to the startCharacter option, except that it applies to containers instead of characters. The next word following this option specifies the first container from which to start searching. The word should be an integer giving the item number of the container in the list of containers passed to Research. For instance,

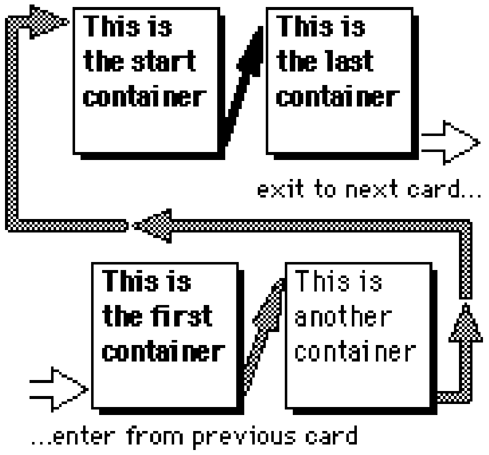
```
research(findMe, "field 1,field 2,field 3", "containers
startContainer 2")
```

sets the start container to "field 2".

How the search proceeds

The search proceeds from the start container, up to the end of the input, and then wraps around to the start of the input and continues searching until the start container is reached again. The following figure illustrates this process. The black arrows show the first phase of the search (during which we search from the start container through the last container, and then continue on to the remainder of the search) and the gray arrows show the second phase of the search (during which we search from the first container, and then reenter the start container). This figure assumes that we're searching more than one card, as described in the following section. If we were searching only the current card, which is the default, then the arrow exiting the last container would wrap around to the arrow entering the first container.

Figure 3. **Searching many containers**


This is the start container
This is the last container
exit to next card...
This is the first container
This is another container
...enter from previous card

StartCard, FirstCard, and LastCard options

The startCard, firstCard, and lastCard options allow you to search through a list of containers in more than one card. With each of these options, the next word in the options list is a number specifying, respectively, the card from which to start searching, the first card to search, and the last card to search. Any one of these options may be omitted, and if so, it will be assigned a default value, as shown in the following table:

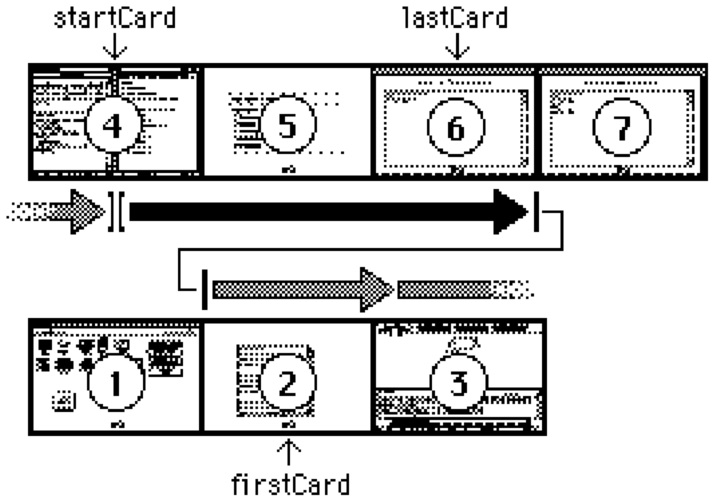
Table 2. **Defaults for startCard, firstCard, and lastCard**

Option	Default	Equivalent HyperTalk expression
startCard	current card	number(this card)
firstCard	1	1
lastCard	cards in stack	number(cards)

How the search proceeds

The search starts from the start card, proceeds through the last card, then wraps around to the first card, and finally continues up to the start card. The following figure should help clarify this process. As before, the black arrows show the first phase of the search (during which we search from card 4 through card 6) and the gray arrows show the second phase of the search (during which we search from card 2 up to card 4).

Figure 4. **Searching many cards**



How Research gets the contents of each container

To get the contents of each container, Research must append the string " of card *cardNumber*" to each container name, where *cardNumber* is replaced by the number of the card being searched. For instance, if the container's name is

```
"script"
```

then, when searching card number 3, the string "script of card 3" is appended, resulting in the name

```
"script of card 3"
```

Similarly, if the container's name is

```
"script of bkgnd btn 3"
```

then the resulting container, when searching card 3, will be

```
"script of bkgnd btn 3 of card 3"
```

Research tells HyperCard to evaluate the expression formed by the name of the container, and HyperCard then returns the contents of the container to Research. If HyperCard could not evaluate the expression, then Research returns the error code ERR_BASIC_HYPERCARD.

Some Examples

The following examples show how the startCard, firstCard, and lastCard options may be used.

The following command

```
research("person", "field 1", "containers¬
startCard 4 firstCard 3 lastCard 5")
```

searches the container "field 1" in cards 4 through 5, and then searches the same container in card 3. The search is actually done on the values of the expressions "field 1 of card 4", "field 1 of card 5", and "field 1 of card 3".

The following command

```
research("person", "field 1", "containers firstCard 3")
```

searches the container "field 1" in cards 3 through the last card in the stack. Assuming that the we're currently in card 5, then we first search from card 5 through the last card in the stack, and then we search cards 3 and 4.

Putting it all together

The options startContainer, startCard, firstCard, and lastCard are used only with the containers option. The startCharacter option is usually also used with the containers option, but can be used on a regular input string. All these options were designed to function together smoothly. While reading the paragraphs that follow, please keep in mind the illustrations showing how Research searches a stack, a list of containers, and a single container.

Each option has a default value, so that when none of these options is specified when you call Research, the default behavior is to simply search the contents of the input string from beginning to end. When the startCharacter option is used, the search wraps around to the start of the input string, in a manner similar to that of most text editors and word processors.

Research extends this wrap-around method of searching to deal with HyperCard's extensions to simple text: multiple fields, containers, and cards. Thus, when the startContainer option is also used, the search starts from the start character in the start container, proceeds up to the end of the start container, and then goes on to the first character of the next container in the list. When the last container in the list is encountered, we loop back to the first container (which may be different from the start container), and search the entire contents of the subsequent containers, until we again reach the start container. Finally, we search from the first character of the start container up to the start character of the start container.

Finally, the startCard option is used to specify the starting point in the hierarchy of objects. We search from the start card in a manner similar to that described in the previous paragraph. The only difference is that instead of immediately reentering the first container on the same card, we instead advance to the first container on the next card. We continue searching up to the last card, and then wrap around and search from the first card up to the start card. Once we again reach the start card, we reenter the first container, and proceed to search up to the start container, as described in the previous paragraph.

Replacing HyperCard's find command

Why I went to such lengths to give Research the flexibility described in the preceding paragraphs will become apparent when one considers the problem of replacing HyperCard's find command. As you probably know, the find command starts from the current location in the stack and searches every following field, in the current card and in subsequent cards, until it either finds a match, or it has searched the entire stack.

Research can be made to do something very similar to the find command. We use the startCard, startContainer, and startCharacter options to specify the current location in the stack, and the chunk option to get the location of the found text. The following command could serve as the basis for a more powerful substitute to HyperCard's find command:

```
research(findMe, "field peopleField, field addressField, card field
commentsField",
      "firstCard" && number(first card of bkgnd peopleBkgnd) &&
      "lastCard" && number(last card of bkgnd
      peopleBkgnd) &&
      "startCard" && number(this card) &&
      "startContainer" && currentFieldNum &&
      "startCharacter" && currentCharacter &&
      "chunk nometa first")
```

In this command, we can get the current field and character from the "selected" operations of HyperTalk (such as selectedChunk). We use the first option to limit the search to the first successful match, and we use the firstCard and lastCard options to limit the search to the cards of our list of people. The chunk expression returned by Research can be used to jump to the card and field containing the found text. If we remove the nometa option, then we can search the fields using regular expressions.

Count

Suppresses normal output and returns instead a count of the number of lines that matched the pattern.

First

The search terminates as soon as the first match is found.

FirstCard

See description of the containers option.

Global

Finds all unique occurrences of the pattern in the line, instead of just the first occurrence (the default). This option automatically turns on the chunk option.

Ignorecase

Ignores character case in comparisons, so that lower case letters match upper case letters and vice versa. Unfortunately, characters with diacritics are still differentiated, so that å is not the same as Å, even though a is the same as A.

Invert

Inverts the search, so that only lines not matching the pattern are found.

Match

Returns true if there was at least one successful match, otherwise returns false. The following script demonstrates one situation where this option could be useful:

Script 2. **GetUserName**

```
-- Ask for user's name.
-- The script accepts the abbreviations "ari",
-- "halberstadt", or "aih" for the name "Ari
-- Halberstadt".
on getUserName
      ask "What is your name?"
      put research("ari|halberstadt|aih", it, "match¬
      ignorecase") into match
      if (match = true) then
            put "Ari Halberstadt" into userName
      else
            put it into userName
      end if
end getUserName
```

NoBusyCursor

Turns off display of the rotating beach-ball while searching. Normally, the beach-ball only appears if the search takes longer than half a second. This option is useful for longer searches in stacks where the rotating beach-ball is inappropriate.

NoMeta

All metacharacters are ignored in the pattern parameter, so that it is not interpreted as a regular expression. This option causes Research to use a very efficient fixed string searching algorithm called the Boyer-Moore-Gosper string searching algorithm, resulting in much faster searches than those possible with full regular expressions.

NoPeriodic

Turns off all periodic actions, such as updating the rotating beach-ball and checking for Command-period events. This could help speed up some searches.

NoUserCancel

Doesn't allow the user to cancel a search by pressing Command-period.

Number

Precedes each output line with the line number followed by a comma. For instance, if the pattern matched line 27, then the output would be:

```
"27,this line was matched"
```

This option has no effect when used with the chunk option.

Separator

Specifies a different character to use as a separator instead of the default return character. The first character of the word following the option is used as the separator character. If no word follows, then the entire input is treated as one giant string. To use a space or a tab as the separator precede the character with a backslash. For instance,

```
"separator ,"
```

will use a comma as a separator, while

```
"separator \" & space
```

will use a space as the separator. Remember that no regular expression matches the separator character.

Changing the separator character from the default has several effects. First, the ^ and $ metacharacters don't match the beginning and end of a line, instead they match the beginning and end of the current unit of text. Second, when output is in chunk form (specified using the chunk option), the chunk expressions use character offsets from the start of the input text, instead of offsets from the start of the matched line. Finally, the number and count options no longer count lines, instead they count input units.

StartCard

See description of the containers option.

StartCharacter

See description of the containers option.

StartContainer

See description of the `containers` option.

UnixMeta

Uses UN*X style metacharacters. This is the default.

Function descriptions

Functions

This section contains an alphabetical list of all of the functions implemented.

"!"

Syntax

```
string research("!")
```

Description

Returns a string giving the version of Research, the full name of the program, the author, a copyright notice, and the date and time of compilation. The string has the basic form “Version 0.9, Research XFCN, by Ari Halberstadt, Copyright © 1990, date time”.

Examples

```
get research("!")
```

"?"

Syntax

```
string research("?")
```

Description

Returns a string giving a brief summary of the functions and call syntax for Research.

Examples

```
get research("?")
```

Error

Syntax

```
error research(Error)
```

Description

Returns the number of the error set by the last function executed. If the last function was executed successfully then returns empty.

Examples

```
get research(error)
```

Matched

Syntax

```
Boolean research(Matched)
```

Description

Returns true if the previous search found at least one match, otherwise returns false.

Examples

```
get research(matched)
```

Searching

Syntax

`string research(`*pattern*`, `*input*`[, `*options*`])`

Description

Searches for the *pattern* in the *input*. Returns lines containing matches, or empty if nothing was found. If any *options* are given then this function will behave differently (see descriptions of options above).

Examples

```
get research(if, script of card 1, invert nometa)
        -- returns all lines not containing the word if
get research("^ *then *$", script of card 1)
        -- returns all lines with only the word "then"
```

Limitations and bugs

This section describes any limitations on the size and number of data that the program may manipulate. Also discussed are any known bugs, with suggested ways to work around them.

Limitations

This section lists various minimum and maximum sizes for Research. All limits may be smaller depending on the availability of memory and other computer resources. It is unlikely Researchwill actually encounter an error associated with the exhaustion of available memory since HyperCard is more likely to quit first.

In the following table, the value represented by Integer is 32,767 and the value represented by LongInt is 2,147,483,647.

Table 4. **Research limits**

Item	Limit
Length of input	LongInt (HyperCard can only pass the first 32000 characters to Research)
Length of output	LongInt (HyperCard can only store the first 32000 characters)
Closure repetitions	255
Expression size	255 characters (less depending on complexity)
Plain pattern size	LongInt (when `nometa` option is used)
Container name	255 characters (less if card numbers must be appended to the name)

Known bugs

This section is included for updates on possible and real bugs, and for the dissemination of temporary solutions. Pseudo-bugs will also be reported here (a pseudo-bug is defined as "weird behavior deriving from the correct definition of the software").

- Though the program appears to work correctly, I am still not convinced that the regular expression compiler always produces a correct NFA (non-deterministic finite state automata).
- The regular expression algorithm I use is fairly slow. I intend to incorporate a better regular expression compiler and a faster matching

algorithm in a future release. I will probably use the Free Software Foundation's GNU grep programs.

Version information

This section describes features that have changed from previous versions. Also discussed are plans for the future of this program.

Changes from earlier versions

There have been no earlier versions.

Future plans

I intend to use a more efficient algorithm for regular expression pattern matching, and perhaps to add a regular expression evaluator for metacharacters used by MPW. I will probably move the regular expression code into a separate code resource which may be loaded as needed (thus MPW metacharacters can be used simply by substituting a different resource). I also may add a capability to search for more than one expression (or at least more than one fixed pattern) at a time.

About the program

«SECTION NOT YET AVAILABLE»

Appendix A. Quick reference

The following table is an alphabetic list of all of the functions implemented, the types of data they return, and their syntax.

Table 5. **Functions**

Returns	Syntax
`string`	`research("!")`
`string`	`research("?")`
`error`	`research(Error)`
`Boolean`	`research(Matched)`
`string`	`research(`*`pattern`*`, `*`input`*`[, `*`options`*`])`

The following table lists all of Research's options, along with a short description of each option.

Table 6. **Options**

Option	Description
Containers	The input string is interpreted as a comma separated list of containers. You can use the startCard, firstCard, and lastCard options to specify a range of cards to search. The startContainer option is useful when replacing HyperTalk's find command.
Chunk	Output is in an abbreviated chunk format.
Count	Counts number of matches.
First	Search terminates after first successful match.
Global	Every unique occurrence is found, instead of just the first occurrence. Automatically turns on the chunk option.
Ignorecase	Ignores upper/lower case distinctions when comparing letters. Has no effect on letters with diacritic marks.
Invert	Finds lines not matching the pattern.
Match	Returns true if there was at least one successful match.
NoBusyCursor	Suppresses display of the rotating beach-ball.
NoMeta	Doesn't interpret metacharacters. Significantly faster (but more limited) than full regular expressions.
NoUserCancel	Doesn't allow the user to cancel using Command-period.

NoPeriodic	Suppresses all periodic actions.
Number	Numbers output lines.
Separator	Next word specifies a different separator character to use, instead of the default return character.
StartCharacter	Starts searching from the given character offset, and then wraps around to the start of the input and continues searching up to the character offset. Especially useful when replacing HyperTalk's find command.
UnixMeta	Uses UN*X style metacharacters. This is the default.

Appendix B. Metacharacters

The table below summarizes all of the metacharacters used.

Table 7. **Metacharacters**

c	any non-special character c matches itself
\c	turn off any special meaning of character c, unless c is **(**, **)**, or a digit
^	beginning of line
$	end of line
.	any single character
[…]	any one of characters in …; ranges like a-z are legal
[^…]	any one of characters not in …; ranges like a-z are legal
\n	when the n'th \(…\) matched (n is a digit from 1 through 9)
r{n}	repeat previous expression n times
r{n,m}	repeat previous expression a minimum of n times and a maximum of m times
r*	zero or more occurrences of r
r+	one or more occurrences of r
r?	zero or one occurrences of r
r1r2	r1 followed by r2
r1|r2	r1 or r2
\(r\)	tagged regular expression; can be nested
(r)	regular expression r; can be nested

Appendix C. Resources used

This appendix gives a complete list of resources needed by this program. These resources must be installed in your stack for the program to work (see the section on installation in the common manual).

The following table lists the resources with their default IDs and names, along with a short description of the data contained in each resource and how the resource is used by the program. The resource of type TABL is described in the common manual.

Table 8. **Resources**

Type	Name	Description
XFCN	Research	The little XFCN whose purpose it is to load, lock, and call the PROC resource.
PROC	Research	The resource containing the executable code.
STR#	Research:ResourceMap	Map of resources used by BinaryTree.
STR#	Research:Info	Version and usage information.
TABL	Research:Functions	The names of the functions.
TABL	Research:Options	The names of the options.
CURS	Research:Busy:0	Four cursors are used for the rotating beach-ball which is displayed while searching.

Appendix D. Revision history

This section is to be used for recording any changes made to this manual. This is necessary since I do not want inconsistencies or mistakes introduced by others to reflect on my reputation, and, if the revisions improve this product, then the person who made the improvements should receive full credit. For consistency, please enter dates as Year-Month-Day.

Table 5. **Revision history**

Date	Name	Comments
90-07-18	Ari Halberstadt	This is an example entry
90-07-11	Ari Halberstadt	Version 0.9