

# **The Blacksmith™ Pattern-Recognition System**

## **Overview & Tutorial**

A review of the pattern-recognition capabilities  
of the Blacksmith™ screen-scraping tool  
from CEL Corporation

Ron Conescu  
Information Management Inc.  
April 7, 1995

Written for  
AirTouch Cellular's  
Poindexter project

# Contents

---

<b>Introduction.....</b>	<b>1</b>	<i>Why this document?</i>
<b>About Blacksmith .....</b>	<b>2</b>	<i>When and why we use screen-scrapers, and Blacksmith in particular</i>
<b>Patterns .....</b>	<b>4</b>	<i>The basic element of Blacksmith's screen-recognition features</i>
Searching for Text .....	4	
Searching for the Cursor.....	5	
Pattern Types .....	5	
Simple Patterns.....	5	
Multiple Targets .....	5	
Compound Patterns.....	5	
<b>Pattern Schemes .....</b>	<b>7</b>	<i>Groups of patterns for complex searches</i>
Component 1: the Search Array .....	7	
Component 2: the Index .....	7	
Component 3: the Version Number .....	9	
Example: Constructing Schemes.....	9	
<b>Regions .....</b>	<b>12</b>	<i>Where should Blacksmith look for a particular pattern?</i>
Searching for Text .....	12	
Searching for the Cursor.....	14	
<b>Help! .....</b>	<b>16</b>	<i>Names and numbers in case of emergency</i>

# Introduction

---

*Why this document?*

I wrote this document because when I started learning Blacksmith, I had difficulty understanding how to use Blacksmith's powerful, flexible pattern-recognition system, which is based on arranging and manipulating numeric codes in various ways.

This document summarizes what I have learned about Blacksmith from experience. It describes *most* of Blacksmith's functionality, not all. It is my hope, however, that after becoming familiar with the way Blacksmith works — after reading this document, and some experimentation on your own — you should be able to:

- rapidly deploy Blacksmith patterns in your own software; and
- refer to the complete, more-technical, official Blacksmith documentation to fill in any missing details, without getting lost.

# About Blacksmith

---

*When and why we use screen-scrapers, and Blacksmith in particular*

Certain types of computers — mainframes, VAXes, etc. — communicate with the world primarily in one way: they display English (or French or Russian) text on a computer screen. The user, typically human, reads the display, and talks back by typing into certain areas of the screen and hitting the Enter key (or any of various other keys to tell the computer “Ok, I’m done typing now.”)

Blacksmith is a **screen-scrafer**, a program for talking to such computers through a **terminal session**, a virtual text-only computer screen. Blacksmith provides commands you can use from your favorite programming language to create a connection to a host computer, to read, write, and check for information in a terminal session, send commands to the host, and programmatically “notice” when the results arrive. It helps you make your computer program pretend to be human — kind of — to “look” at the terminal window and see what’s there.

The commands for reading and writing are straightforward: you specify a screen location, and Blacksmith will either return the information there or put your text there, as appropriate.

Checking for information, however, is more complex. Terminal windows are intended to be read by humans, not computer programs. Thus, the information that appears in the windows can vary tremendously. Computer programs trying to “read” terminal windows need to look for very specific pieces of data in order to “recognize” when a certain result has appeared.

The simplest type of information to look for is a particular string in a very specific location. Often, however, you don’t know specifically where your information will appear, and frequently you don’t even know what you’re looking for — just that *something* should be there.

The simplest screen-scraping programs allow you to check for a certain string in a certain position on the screen. Blacksmith allows you to do that, but it is also much more flexible. For example, given the window below, you could ask Blacksmith to check for the following:

```

Poindexter's Firestone Window

THIRD PARTY CREDIT BUREAU REPORTING (RCBRED)

PLS REENTER... DB2A, ZIPCD,CRBUR,

CLIENT ??????  AGENT 2000229  ACCOUNT NUMBER  -
LAST NAME SNOW  FIRST NAME BETSY  M.I.
STREET 111 W JACKSON  CITY ATLANTA  STATE GA  ZIP 30315
BDATE 01 / 01 / 60  SS# 341 - 39 - 2992  PH# 404 - 555 - 1212
EMP HERE  YRS 05  OPR 404  WK# 555 - 121 - 2123
DR# 4567  SALARY K
CR BUR  SCORE CODE
EXISTING SERVICE N NO OF PHONES 01 PHONE/ACCT
** ERROR ==> AGENT NOT FOUND ON DB2 DATABASE CONTACT SUPERVISOR TO AUTHORIZE

:02.6 7,25

```

- Is row 5 blank? (No.)
- Is the first error code on line 5 any of DB2A, STATE, or CRBUR? (Yes.)
- Is there a ?????? anywhere on the screen? (Yes.)
- Is there a ?????? at the current cursor location? (No.)
- Is the word ERROR at location (15,5)? Note that in Blacksmith, the row coordinate is given first; that's row 15, column 5. (Yes.)
- Are we looking at a blank copy of this screen? Specifically: Is the text CREDIT BUREAU in row 2, the cursor at (7,25), and has the text at that location been cleared? (No.)

# Patterns

---

*The basic element of Blacksmith's screen-recognition features*

Blacksmith, thus, looks for **patterns**, not simply strings of characters. A pattern specifies a **region** of the screen in which to search for any of several **targets**.

A region is typically one of the following:

- anywhere on the screen
- on a certain row
- position and length: a certain range of columns on a certain row
- rectangular area: from coordinates (top, left) to (bottom, right)

A target is one of the following:

- a text string
- the cursor

## Searching for Text

You can look for the following types of text:

- a specific string, optionally case-sensitive (e.g., Supervisor)
- something *other* than a specific string, optionally case-sensitive (something other than Supervisor)
- a blank (no text)
- a non-blank (any text at all)

... in these locations:

- anywhere on the screen
- anywhere on a row (very common)
- position and length: a certain range of columns on a certain row
- within a rectangular area: e.g., columns 1-10 of rows 20-24

Furthermore, when you specify the coordinates of one of these locations, you can also specify whether that coordinate is *absolute*, an actual screen location, or *relative* to the cursor's current position. Examples (from the screen above):

- Is there a ?????? at the current cursor location? (No.)
- Does the word ERROR appear in column 5 (absolute), 8 rows below the cursor's current location (relative)? (Yes.)

## Searching for the Cursor

You can also look for the cursor itself, as follows:

- a specific position (row, col)
- in a specific row
- in a specific column
- at a specific row and column, within a specific number of characters
- within a rectangular area: e.g., columns 1-10 of rows 20-24

## Pattern Types

### Simple Patterns

A pattern is composed of:

- **target definition:** what are we looking for?
- **region definition:** where do we look?

and looks like this:

*region*•*target*

Blacksmith will look in *region* for *target* . It returns 1 (true) if it finds *target*, 0 (false) if it doesn't.

### Multiple Targets

If a pattern contains more than one target, like

*region*•A•B•C

Blacksmith will look in *region* for each target. If it finds any of them, it tells you which (1, 2 or 3); if it finds none, it returns 0.

## Compound Patterns

You can also tell Blacksmith to look for more than one pattern at a time. For example:

```
region 1•A•region 2•B•C
```

That is:

```
pattern 1 = region 1•A  
pattern 2 = region 2•B•C
```

Blacksmith will look for *both* patterns; it will return 1 (true) if it finds them both, and 0 (false) if it finds one or the other, or neither, but *not* both.

# Pattern Schemes

---

*Groups of patterns for complex searches*

You instruct Blacksmith to search for patterns using a **scheme**.<sup>1</sup> A scheme has the following components:

- the **search array**, containing the patterns you want to search for
- an **index** (in the back-of-the-book sense of the word) for the search array
- the current **version** of the Blacksmith pattern-recognition system

The components are delimited by a **separator character** of your choice, such as a bullet (‘•’, which is option-8 on the Macintosh keyboard). A scheme looks like this:

```
version•index•search array
```

## Component 1: the Search Array

A search array is composed of a series of patterns, as discussed above:

```
pattern 1•pattern 2•...•pattern n
```

Remember, of course, that a pattern is composed of a **region definition** and a series of **target definitions**:

```
region•target 1•target 2•...•target n
```

## Component 2: the Index

It’s easy to build complex search arrays: lots of patterns, each having several targets. Thus, it can get kind of messy in there...

To clarify where each pattern in the search array starts and stops, you make a map: the **index**. The index contains the *location of each pattern in the scheme*. Blacksmith uses the index to know how many patterns you have specified and where each one begins.

---

<sup>1</sup> Actually, CEL Corporation, the makers of Blacksmith, doesn’t call it a “scheme”; they call it a Blacksmith Pattern Definition, or BPD. However, I thought the acronym would hamper clarity, so I used my own word...

### Simplest case: one pattern

The simplest example of a scheme is

```
version•index•pattern
```

To come up with the index for this scheme, we look at the positions of the scheme components:

<b>position</b>	<b>what's there</b>
1	version
2	index
3	pattern

There's only one pattern here, so the index is 3: the pattern's position in the scheme.

```
version•3•pattern
```

### Slightly tougher: two patterns

With more than one pattern, designing the index is more complicated, because a pattern itself has several components. For example:

```
version•index•pattern 1•pattern 2
```

If pattern 1 has two targets, and pattern 2 has one target, then the full scheme is:

```
version•index•region 1•target 1a•target 1b•region 2•target 2
```

The positions here are:

<b>position</b>	<b>what's there</b>
1	version
2	index
3	region 1
4	target 1 a
5	target 1 b
6	region 2
7	target 2

The index lists the starting positions of each pattern. Since pattern 1 starts at position 3 and pattern 2 starts at position 6, the index is

```
3,6
```

and the entire scheme is

```
version•3,6•region 1•target 1a•target 1b•region 2•target 2
```

# Component 3: the Version Number

The version number is always 1.

The brain of Blacksmith, the engine which understands and searches for patterns, is known as the Blacksmith Pattern-Matching Facility (PMF). Currently, only version 1 of the PMF is supported.

You can think of it as a convenient way to specify the separator character; the PMF knows that whatever follows that first number in the scheme is the separator.

## Example: Constructing Schemes

By now you've learned how to construct **patterns**, **schemes** of patterns, and **indexes**. It's about time for an example to sum things up so far.

- Does this screen have an error about a person named Betsy who lives in either Atlanta or Macon? (Yes)

We need to look for the following patterns:

- search line 8 for the name "Betsy"
- search line 9 for either "Atlanta" or "Macon"
- search line 15 for the word "error"

```
Poindexter's Firestone Window
THIRD PARTY CREDIT BUREAU REPORTING (RCBRED)
PLS REENTER... IB2A, ZIPCD, CRBUR,
CLIENT ?????? AGENT 2000229 ACCOUNT NUMBER -
LAST NAME SNOW FIRST NAME BETSY M I
STREET 111 W JACKSON CITY ATLANTA STATE GA ZIP 30315
BDATE 01 / 01 / 60 SS# 341 - 39 - 2992 PH# 404 - 555 - 1212
EMP HERE YRS 05 OPR 404 WK# 555 - 121 - 2123
DR# 4567 SALARY K
CR BUR SCORE CODE
EXISTING SERVICE N NO OF PHONES 01 PHONE/ACCT
** ERROR ==> AGENT NOT FOUND ON IB2 DATABASE CONTACT SUPERVISOR TO AUTHORIZE
:02.6 7,25
```

In Blacksmith-ese, that translates to:

```
pattern 1: 8•Betsy region•target
pattern 2: 9•Atlanta•Macon region•target 1•target 2
pattern 3: 15•error region•target
```

OK. Here's a blank scheme:

```
version•index•search array
```

The version number is always 1, so we start out with

```
1•index•search array
```

Now, let's do the search array:

```
search array = pattern 1•pattern 2•pattern 3
```

Fill in the actual patterns:

8•Betsy•9•Atlanta•Macon•15•error

Now, here's the problem: *you* know there are three patterns here. Blacksmith, however, looking at this string of text, sees the command: "look on row 8 for any of the following strings: 'Betsy', '9', 'Atlanta', 'Macon', '15', and 'error'."

Without the aid of an index, Blacksmith would return True — "Yes, I found a match" — because the string "Betsy" is, indeed, on line 8. It would never check for the other patterns; it wouldn't realize there were any more.

So let's construct the index. First, place the search array in the scheme to check the real positions:

1•index•8•Betsy•9•Atlanta•Macon•15•error

The positions here are:

<u>position</u>	<u>what's there</u>	<u>type</u>
1	1	version
2	(index)	index
3	8	<b>region 1</b>
4	Betsy	target 1
5	9	<b>region 2</b>
6	Atlanta	target 2 a
7	Macon	target 2 b
8	15	<b>region 3</b>
9	error	target 3

Since the index lists the starting positions of each pattern, the index would be

3, 5, 8

And, at long last, we have the scheme:

1•3, 5, 8•8•Betsy•9•Atlanta•Macon•15•error

Blacksmith looks at the index and says to itself: "Oooh, fun! A search scheme! OK: I have 3 patterns to find. The first starts at position 3, the second at position 5, and the third at position 8. If I can't find 'em all, I'll tell the user that the search failed. If I *do* find 'em all, though, I'll say it succeeded."

The search, in this case, would succeed.

# Regions

---

*Where should Blacksmith look for a particular pattern?*

## Searching for Text

The basic pattern for text searches is:

```
region•target
```

### Region Types

The region itself has two components:

- **location:** where to look for the text
- **type:** how to look for the text

The most common locations to search for text — anywhere on the screen, and anywhere on a specified row — are easy to specify:

```
0•error      location = 0: looks for the word “error” anywhere on the screen
```

```
15•error     location = row number: looks for the word “error” anywhere on  
line 15
```

It is also possible to restrict the location: to search in a certain field on the screen, or within a certain rectangular area. The following pattern looks for the word “error” on row 15, in the 5 characters starting in column 6:

```
4,15,6,5•error      type = 4; location = at (15,6) for 5 characters
```

Note that the type has been tacked onto the front of the region definition; it is separated from the location by a comma, not a bullet.

The following looks for the word “error” in any of the last three rows on an 80x24 character screen:

```
5,22,1,24,80•error  type = 5; location = from (22,1) to (24,80)
```

You might ask: so what good does a type do? Why not simply specify the coordinates I want to search? (I asked that very question for months...) The answer: once you’ve restricted the area, Blacksmith also allows you to hone the search in other ways...

## Text Constraints

If you're using region types, then you can also add the following criteria to the search string:

- case sensitive ('yoda' ≠ 'Yoda' ≠ 'YODA')
- Boolean search: see if something *other* than the specified string was found

To add these criteria, add the appropriate **modifier** to the **region type**:

	Case Sensitive?	
	no	yes
string found	0	4
string not found	2	6

Recall the following example from above:

4,15,6,5•error                    type = 4; location = at (15,6) for 5 characters

This pattern will return true if it finds 'error', 'ERROR', 'Error', etc. To ensure that it only returns true if it finds the lower-case 'error', we add 4 (from the chart) to the region type:

8,15,6,5•error                    type = 4, modifier = 4

To make the pattern return true if something *other* than a lower-case "error" is in that position, you add 6, not 4, to the type:

10,15,6,5•error                    type = 4, modifier = 6

And, finally, to see if something other than the word "error" (regardless of case) is at that location, add 2 to the type:

6,15,6,5•error                    type = 4, modifier = 2

## Searching for Emptiness

Conversely, you can test for a blank (or a non-blank) at a particular location. Recall another example from above, using type = 5:

5,22,1,24,80•error                type = 5; location = from (22,1) to (24,80)

That looks for the word "error" anywhere in the last 3 lines of the screen. How 'bout seeing if those three lines are empty?

13,22,1,24,80                    type = 5; modifier = 8; no target

This pattern consists entirely of the region definition; it has no target. It can be hugely useful to test if an error-message area contains *nothing*, i.e., everything's OK with that screen. Similarly, you could see if there were anything at all in that region:

15,22,1,24,80                      type = 5; modifier = 10

### Cursor-Relative Searches

On top of all that, you can also tell Blacksmith that the specified coordinates are relative to the current cursor location, instead of relative to the upper-left corner of the screen (i.e., absolute). To do this, add one of the following modifiers to the region type:

Relative to:	Add:
(0,0)	0
cursor row	100
cursor column	200
both	300

### Exercises

Try these examples, from the terminal window on page 2:

- Are there any errors on this screen?
- Is the City field full?

## Searching for the Cursor

Cursor-search patterns — and other non-textual search patterns<sup>2</sup> — have the following format:

prefix•region

For cursor searches, the prefix is always -1:

-1•region

---

<sup>2</sup> Another such is a field-count search: Blacksmith allows you to check the number of fields in the terminal window, and gather some statistics about them. However, since I have not used this feature, I won't discuss it. :-)

The region, as for text searches, consists of a **type** and a **location**:

<b>Type</b>	<b>Location</b>	<b>Blacksmith searches for the cursor...</b>
1	offset	at the position which is offset characters from (0,0), counting left to right and top to bottom
2	row	in row
3	column	in column
4	row, col, len	starting at (row, col) for len
5	top, left, bottom, right	within that rectangle

So, for example, the pattern

`-1•4, 7, 15, 1`

type = 4; starting at (7,15) for 1 character

looks in position (7,15) for the cursor.

# Help!

---

*Names and numbers in case of emergency*

## **CEL Corporation (makers of Blacksmith)**

P.O. Box 8339  
Edmonton, Alberta T6H 4W6  
CANADA  
Voice: (403) 463-9090  
Fax: (403) 428-5376

AppleLink: CEL.MKTG  
CompuServe: 76060,152  
AOL: CEL Soft

Blacksmith, and all its components, ©1991-present by CEL Corporation.

## **Information Management Incorporated**

One Decatur Towncenter  
150 E. Ponce de Leon Avenue  
Decatur, GA 30030  
Voice: (404) 377-4840  
Fax: (404) 377-5116

Curtis Crowson: x304      Curtis\_Crowson@infoman.com  
Ron Conescu: x319      Ron\_Conescu@infoman.com