The CRM114 Discriminator Revealed!

-or-

How I Learned to Stop Worrying and Love My Automatic Monitoring Systems

to match CRM114 Version 20061005

-by-

William S. Yerazunis, PhD

Copyright Information

Colophon
This work was produced on Fujitsu P2120 and Dell m700 mini laptops, running Red Hat Linux and OpenOffice, mostly on the Boston MBTA 6:53 AM train to Cambridge . Neither proprietary software nor caffeine was used.

What is this? Some kind of grep bitten by a radioactive spider?

– apocryphal

Introduction

By the fact that you're reading this book, it's probably true that you're interested in using the CRM114 Discriminator (or 'crm114' for short). You're probably thinking of using it for filtering and sorting email, and probably hoping that it's not too hard to learn.

Hopefully, you're right on all three counts, but it's a mistake to think of CRM114 as <u>only</u> a spam filter. CRM114 is a Turing-complete[1] programming language, optimized to write filters for “difficult problems” like text classification. It just happens that spam filtering is a good application for CRM114.

It wouldn't be fair not to warn you that CRM114 contains a few mind-bending constructs, and we certainly wouldn't recommend it as a first programming language. But, if you need to do very fast (say, soft real-time) filtering of data streams when the problem itself is poorly defined, then CRM114 may be just the ticket.

Grab the red pills and a tall glass of ice water, and dig in...

– Bill Yerazunis

1 Well, modulo that your memory is limited to the address space of the hardware.

“... Now, in order to prevent the enemy from issuing fake or confusing orders, the CRM114 Discriminator is designed not to receive *at all*... That is, not unless the message is preceded by the proper three-letter code group.“

– George C. Scott, playing the role of General Buck Turgidson, in Stanley Kubrick's **Dr. Strangelove**

Table of Contents

A Few Conventions

As usual in most computer books, different fonts are used to describe things about the text. In this book, we use several different font styles to make our meanings clear.

- Text in this font means "you should read this". Most of this document is set in this font.
- **`Text in this font`** means **`"you typed this"`** This is what's used for programs and data that you input.
- `Text in this font` means "`the computer typed this back at you".` This is what's used for computer listings or output.
- *`Text in this font`* means *`"you should substitute your own text here".`* This is what's used where you should put something appropriate to your program – that is, a *metasyntactic* variable. This is often an argument name.
- *Text in this font* means "*something that happens that isn't printable text*" This is used for non-printing characters that aren't obvious from context, or when the interaction is "off screen", like interaction with another process or program or via the net.

- Text in this font with a floating box are advanced notes for people who want to know "why". Think of this as a fireside chat with the language builders. You might not agree with the notes, but at least you'll know why something is the way it is, and what kind of argument might convince the language builders to change it to be the way you want it.

All of the examples in this book are actual cut-and-pasted text from running CRM114 sessions, so with any luck, there should be very few buggy examples.

A few useful things to know about CRM114

1. CRM114 is a sequential language. That is, statements are executed one after another (unlike, say, Prolog). Running off the end of the program is an implicit "exit". Running off the end of the main routine into a subroutine is not an exit; execution continues into the subroutine.

2. CRM114 has one basic data storage structure – the string. There's no such thing as an object, or a floating point number, or a file descriptor. It's strings "all the way down". [2]

3. CRM114 strings can overlap. If two strings overlap, and you change something in the overlapped region, both strings change. That includes string length – if you change the length of an inside string, the outside string changes length too.

4. CRM114 is, at heart, a generalized filter, and like its movie namesake, defaults to "no output". Remember this as "When in doubt, nothing out".

5. You never have to worry about string pointers, or memory allocation, or memory leaks; CRM114 keeps track of all of the strings internally. CRM114 has an in-flight reclaimer that uses program structure information to immediately reclaim unused memory with very low impact.

6. CRM114 will never grow memory without bound. Once CRM114 starts up, it will never try to allocate significantly more memory. This makes it easy to "jail" a CRM114 application against denial-of-service (DoS) attacks. You might consider that, at least with respect to memory usage, all CRM114 programs are already in "jail".

7. CRM114 uses a built-in JIT (Just In Time) microcompiler; this makes program startup very inexpensive. The only things that are fully computed before your program starts are the statement types, program labels, and the overall structure so that runtime error traps know where to go.

8. CRM114 is very good at calling other programs or tools, so if you ever think to yourself

2 Long ago, a technologically advanced people thought the world sat on the back of a giant turtle. One young child asked "what does the giant turtle stand on?", His father was unable to answer. His grandfather was unable to answer. The tribal chief was unable to answer. But the medicine man told the child "It's turtles all the way down.". With this, the child was satisfied, and went on to become a great mathematician.

“gee, I can do this with “make”, or “units” or “mysql”, you can. It's easy to fork off another program, as a separate process, send it data, get the results, and clean up afterwards, in just one line of code.

9. CRM114 is also a “good citizen” and is easy to call from other languages at the command level. (Unfortunately, runtime-linked call-in / call-out at the binary level, sometimes called native methods, is not currently in the language).

10. CRM114 is capable of “demonized” operation, where a long-running process forks subprocesses, with each subprocess servicing a single request, either from a network or from a user. An advantage of demonized operation is that the initial JIT compiler computation are now passed to each subprocess, so the JIT does not need to be repeated.

11. CRM114 has some extremely powerful built-in operators for text correlation, matching, and classification. The CRM114 framework makes it very easy to add new classifiers to CRM114. These classifiers are carefully implemented for speed and accuracy and can save a lot of developer time.

How this book is organized

This book is divided into six Sections and three Appendices. These sections are intentionally set up to minimize the number of times you're exposed to a concept without all of the other concepts involved being described and demonstrated first. Whether or not this is successful is an open question; in any case whenever a statement is used that you haven't seen before, we'll provide at least a minimal explanation.

- Section 0 (you're reading it right now) covers the fine print about typographical conventions, what can be found where, etc. Consider Section 0 to be administrative overhead after you've read it once.
- Section 1 shows a few examples of very simple programs that can be intuitively understood, and sets the stage for understanding CRM114.
- Section 2 looks at the basic operations of CRM114 – the concepts of program structures, regexes, var-expansion, and quoting.
- Section 3 does a walk through of all of the statements in CRM114, and shows examples of each statement doing something. This is the gist of CRM114 programming style, and introduces the statements, so the reader can see how the language fits together.
- Section 4 shows how to debug and optimize your CRM114 programs.
- Section 5 shows a batch of very useful idioms to do useful things. This is the "Kinks and Hacks" section.
- Section 6 shows how to build a simple but fully functional anti-spam filter from the ground up.
- Appendix A is the full reference page for every CRM114 statement. This is the full and definitive description of every option, flag, variable, and action available.
- Appendix B explains how to download the CRM114 packages and install the base CRM114 engine.
- Appendix C is the full listing of a moderately effective mailfilter; this is the same filter that you get to build yourself in Section 6.
- The Index finishes out the book; if you already have some inkling of what you need to know, use the Index to jump directly there.

“CRM114 looks a lot like TECO... on acid. And I mean that in the most affectionate way possible.”

-- (or something very similar) Eric Johansson, at LISA 2004

Section 1: Your First CRM114 Program

A very wise man once said that the first program to write in any language is “Hello, world”, and CRM114 is no exception.

First, let's make sure that CRM114 is installed. At the operating system's command prompt, type:

```
crm -v
```

Getting anything back that says something reasonable, like this:

```
# crm -v
This is CRM114, version 20050518.BlameMercury (TRE 0.7.2)
Copyright 2001-2005 William S. Yerazunis
This software is licensed under the GPL with ABSOLUTELY NO WARRANTY
#
```

This is a good sign. If you don't see something similarly encouraging, best to either ask your system administrator to install CRM114 for you, or to skip to Appendix 1 and install CRM114 for yourself.

The first program can be put right on the command line:

```
crm '-{ output /Hello, world!\n/ }'
```

Hit return, then hit EOF (ctrl-D on a Linux system, ctrl-Z on Windows[3]). You should see:

```
# crm '-{ output /Hello, world!\n/ }'
return, then ctrl-D
Hello, world!
#
```

So, your first program worked. Some things to notice:

3 As most CRM114 users are on Linux (or other *nix variant) we'll use ctrl-D for the rest of this manual. Windows users will need to remember that they should use ctrl-Z.

1) You can put a program right on the command line. This is handy for testing out ideas.

2) When you do program from the command line, you will probably need to put single quotes `'like this'` around your program. That's because `bash` and many other command line interpreters will try to execute your program directly, and that won't work because CRM114 programs aren't bash programs.

3) You had to send EOF to your program before it runs. This is the default mode of CRM114 action – as soon as the engine begins execution, it runs just enough of the compiler to know that your program isn't hopelessly garbled; then it reads input data from `stdin` till EOF.

4) We can print out a newline with `\n` as we might expect. (Actually, the full set of ANSI `\`-whatever characters is allowed, including `\t` for tab, and `\a` for a bell).

This default **"read-to-EOF"** is a design decision. There are other ways in CRM114 to read input, but most filtering programs will be perfectly happy to just read it all at the very start of program execution, so that's the default. Some of those other ways include ways to read past end of file (always handy, that) and ways to do random I/O; read-to-EOF is just the default. But we're just starting out, so we will stick with programs that read their entire input from stdin and then begin execution.

Simple Input and Output

Now that we know how to do output, and we know that CRM114 will do input by itself, we can write a simple `cat` program. This program just reads `stdin` and then writes the same data to `stdout`, without changing it in any way.

A good question following from CRM114's "read to EOF" principle is "When CRM114 reads `stdin`, where does it put what it read?" By default, CRM114's initial input is placed in a variable called the *default data window*. The default data window has the name:

```
    :_dw:
```

(mnemonic: Data Window) and it's created automatically during program initialization, so you don't have to declare it in any way. Whenever you tell CRM114 to do something, and you don't tell it what to use for data, it will use the contents of **`:_dw:`**[4] Of course, you are not limited to this one variable; CRM114 can have many variables (there is a default limit, to prevent DoS attacks). There are only a few rules for defining a variable name; we'll get into those in the next section.

Here's a program that uses CRM114's automatic read-to-EOF and then prints everything it read.

```
    crm '-{ output /:*:_dw:/ }'
```

When you type this into the command line, and give it some data, you get:

```
    # crm '-{ output /:*:_dw:/ }'
    this is a test
    this is only a test
    one two three.
    hit return, then ctrl-D for end-of-file
    this is a test
    this is only a test
    one two three.
    #
```

4 If you're a computer-science type, think of the data window as CRM114's Turing tape. Since CRM114 can mutate the symbols on the tape and branch depending on the tape contents, CRM114 is Turing-equivalent.

which is pretty much unsurprising. Let's take this program apart.

- First, the single quotes in '-{ and }' are only there to keep the bash command-line interpreter from stealing the whole program and trying to execute it.
- The "program" itself is just one statement long:

```
                    output /:*:_dw:/
```

- Clearly, "output" is a verb – a command to output something.
- The only question remaining is

    what does **`/:*:_dw:/`** `mean` ?"

Let's take it from the inside out and make some educated guesses. We've already learned that the default data window (the default data to use if you don't specify something else) is called **`:_dw:`** . The starting and ending slashes are probably some kind of quote marks (which is true, the starting and ending slashes are a form of quoting).

That reduces the problem to

What the heck is **`:*`** ?

The **`:*`** (a colon followed by a star) is the *var-expansion operator* – it takes whatever immediately follows (no spaces allowed!) as a variable name, looks up the value of that variable, and splices the value back in. The reason for **`:*`** as the var-expansion operator will become clearer later when we learn about *regex expressions*.

We can now translate this program to English:

1. CRM114 default action – read `stdin` to EOF,
2. CRM114 default action - put the result of that read in **`:_dw:`**
3. First program statement: start executing an OUTPUT statement
4. Output wants the value of **`:*:_dw:`** - the :* means we need to var-expand.
5. look up the variable named **`:_dw:`** , expand it, and splice it into the output string.
6. Output the spliced-up output string to `stdout`.

That's the full interpretation of this program. Once you know what **`:*`** does, it's not complicated. CRM114 has several such operators as **`:*`** to do things like variable var-expansion, string length, math evaluation, etc. We'll visit this subject in more depth under *variable expansion,* or var expansion, for short.

“CRM114 isn't ugly like PERL. It's a whole different *kind* of ugly.”
- John Bowker

Section 2: CRM114 Program Structure, Quoting, and Meaning

In the last chapter, you noticed that CRM114 used slashes "`/`" to denote what we wanted to print, instead of the more common quote marks. CRM114 is rather unique among computer languages in that CRM114 uses different forms of an argument (such as different kinds of quote marks) to mean different things – this is called *declensional* syntax.

Most computer languages use *positional* syntax, not declensional; the meaning of some text depends on where it is in the command, and getting things in the wrong order will cause program bugs[5]. In CRM114, the quoting tells the meaning of the text, and mixing up the order will not change the meaning.

- No quoting at all means the command action name itself – the action must go first in the command (the command name is the only thing that has a fixed position).
- Colons **`:like_this:`** mean a variable. A variable by itself on a line is a *program label* that you can GOTO or CALL (and it can also have a string value).
- Slashes **`/like this/`** are used to indicate a pattern. Often, this subject is a regex (a regular expression, we'll talk about those later). Other times, the pattern is being evaluated by the command (as above `OUTPUT` evaluating a pattern causes the pattern to be output).
- Parenthesis **`(like this)`** are used to indicate things that will be arguments--and often, things that will be changed by the command's execution.
- Angle brackets **`<like this>`** are flags that change the defaults of a command.
- Square brackets (sometimes called brackets or boxes) **`[like this]`** are "where" to do the command; they limit the action of the command to that domain, and they usually are read-only arguments.
- Semicolons (**`;`**) separate statements, as does the unprintable *end-of-line.*

5 If you think positional is good, then, without looking at the man page, what's the argument sequence for `calloc()` ? Now, look it up. Did you get it right?

- Curly braces **`{ like this }`** group multiple lines of commands together into blocks.
- A comment starts with `#` (anywhere in the line) and ends at the end of a line or with **`\#`**
- Notice that the regular quote marks like `'` or `"` or even `` ` `` aren't meaningful to to CRM114; you can use them freely in your text and they won't affect your programming code. (but watch out, a single quote like `'` on the input line may be eaten by **`bash`**. Programs stored in files don't have this problem).

With all of these different kinds of quotes, you might think that CRM114 would be confusing. Actually, it works out very well:

- Each of the quote types -- **`(), [], <>, //`** and **`{}`** -- are used about equally often.
- Each of the quote types always mean the same thing.
- Only the outermost group of quotes "matters" (inside quotes of different types are disregarded).

Once you start to use the language, it becomes easy to figure out what the different quote marks mean. The only issue with this scheme is that filenames that specify a directory with embedded slashes like **`/var/log/messages`** needs to use backslashes to escape the forward slashes, so we usually have to write it as **`\/var\/log\/messages.`** This is only the case for strings delimited by slashes; if the string quoting was a different kind of quote, then you don't have to escape the slashes (though you do have to escape the kind of quote used).

Because the arguments to every kind of command are always quoted in the same methodology, it's easy to figure out what any command does (or at least, how to make a reasonable guess as to what it's supposed to do). This improves readability of the code tremendously.

There's another advantage-- in CRM114, the command name goes first, and everything else (with a few exceptions) can go in any order at all. There's no need to try to remember any particular order[6]. This makes it easy to write pretty code – and pretty code is understandable code- a lesson worth committing to memory no matter what language you program in.

6 Exceptions are rare-- but typically mean "use this, then use that" for commands that will need more than one pattern, or "here's an argument, and here's a different kind of argument". In these cases, we've tried to structure the command semantics so that the mandatory kinds of arguments go first, to give a logical ordering to the arguments. But only five commands in all of CRM114 have any such duplication (they are LEARN, CLASSIFY, WINDOW, TRANSLATE, and SYSCALL).

This use of declensional syntax instead of positional syntax was a grand experiment to see how well it worked in real life. The result is “Hey, it's pretty decent”, neither an unqualified success nor failure.

An upside is that by allowing the coder to put things in whatever order fits best to provide an easily understandable structure, you get code that has a lot more symmetry and hence is easier to check (people *are* good pattern-matchers, after all, and we'd be foolish not to exploit that ability).

The biggest downside is that parser generators such as YACC and BISON weren't really built with declensional parsing in mind, so it's not easy to create a specification file for BISON to create an error-checking parser for CRM114 syntax. As a result, CRM114 contains several hand-carved parsers for various parts of the language. An upside of this downside is that with a hand-carved parser, it is easy to implement the JIT (Just In Time) compiler.

In short, people get along just fine with declensional syntaxes (after all, declension is a human language invention), it’s old-school LALR-1 tools like YACC that .... er... yakk up a hairball when dealing with declensional languages.

The CRM114 Anti-Denial of Service (anti-DoS) Profile

Many of CRM114's design decisions are motivated by a need for programs to remain secure when being fed malicious data. This isn't to say that it's difficult to write a CRM114 program that is malicious – that's easy; CRM114 does not go out of it’s way to protect you from shooting yourself in the foot. Rather, CRM114 was designed so that a CRM114 program that wasn't designed to be malicious should be hard to trick into becoming malicious. Usually, this trickery simply consumes resources and keeps other legitimate users from running their programs or getting their mail (usually called a Denial of Service attack, or DoS for short), so the prevention system is called an anti-Denial-of-Service profile.

For instance, CRM114 loop interation is designed so it's hard to write a data-dependent loop that can become infinite that won't be infinite during testing; iteration indices are not exposed to be mangled but are an intrinsic property of the variable being iterated over. String expansion is designed to NOT be easy to trick by being fed data that contains embedded quotes. Of course, all of the data paths are clean for 8-bit characters; are counted-length (although all are also dynamically resized automatically as needed), and are NULL-safe except where that would violate the POSIX standards for things like filenames.

Finally, CRM114 strings can't grow without bound, thereby taking up all of virtual memory. Instead, total CRM114 string usage is bounded to a predetermined total size. If a program gets handed enough data to exceed that bound, the program gets a fatal error (which it can trap and try to recover from, or just die). This also helps program speed, as CRM114 doesn't constantly malloc and free memory; it's preallocated once at startup, and internally accounted for the rest of program execution.

Variables and Var-Expansion

We've used **`:_dw:`** for a program; let's now consider user-defined variables. CRM114 allows a large number of simultaneous variables, and there are only three rules required.

Rules for Variable Names

Here are the rules for a variable name (or a GOTO label – both must follow the same rules)

1. All variable names and labels start with a colon **`':'`** The variable ends with the next colon **`':'`**

2. Variable names and labels can have names containing only ink-printing characters – but no colons (because according to rule 1 the next colon ends the name).

3. Don't name your own variables or labels with **`:_`** (colon-underscore) as the first two characters. Variables with those two first characters are reserved for the execution engine (and all such variable names will conform with rule 1 and rule 2). Reading such variables is safe but may or may not be meaningful; some (but <u>not</u> all) can be safely altered as well. If you change an important internal variable you might cause the execution engine to fail.

That's pretty much all of the rules required for a variable name.

Now you might understand why **`:*`** is the operator for variable name-to-value substitution. When the **`:*`** is concatenated onto the front of a variable (that is, the **`:`**), it forms **`:*:`** . In the case of a regex, **`:*:`** is the same as **`:+`** , so nobody would ever use **`:*:`** to mean the same thing as **`:+`** . This leaves us free to use **`:*:`** to mean something quite different – variable substitution. Similarly, **`:+:`** as a regex would mean one or more colons, followed by a colon; since **`::+`** equally usable, the **`:+:`** is available to reuse as variable redirection.

Built-in Variables

We mentioned that there are a number of built-in system variables, which are:

- **`:_nl:`** - newline
- **`:_ht:`** - horizontal tab
- **`:_bs:`** - backspace
- **`:_sl:`** - a slash
- **`:_sc:`** - a semicolon
- **`:_cd:`** - call depth – how deep in the call stack is the current routine
- **`:_cs:`** - current statement – what is the current statement number
- **`:_arg0:`** thru **`:_argN:`** - command line args, including <u>all</u> user-specified flags
- **`:_argc:`** - how many command line arguments there were
- **`:_pos0:`** thru **`:_posN:`** - positional args ('-' or '--' args deleted)
- **`:_posc:`** - how many positional arguments there were
- **`:_pos_str:`** - all positional arg uments concatenated
- **`:_env_whatever:`** - environment value 'whatever'
- **`:_env_string:`** - all environmental arguments concatenated
- **`:_crm_version:`** - the version of the CRM system
- **`:_pgm_hash:`** - hash of the current user program, for version verification
- **`:_pid:`** - the PID of the current process
- **`:_ppid:`** - the PID of the parent of current process
- **`:_iso:`** - the current isolated data block (don't change this!!!)
- **`:_dw:`** - the current data window (the default argument for most statements)

Overlapping Variables and How They Interact.

How the variables interact is a little more interesting than the variable naming rules. Remember, a CRM114 variable is not just a string – it's an <u>overlapped</u> string. That is, it may share actual storage space with another string – or with more than one other string.

This gets especially useful (or mind-bending, depending on your philosophy of programming) when you have multiple overlapping strings and you change one of the strings – the other strings change in synchrony, at zero computational cost (actually, almost zero- CRM114 does do some arithmetic on the starts and lengths of each participating string). Even changes of length are done in synchrony – the rule is “pretend to insert or delete enough characters at the start of the string being changed to make exactly enough room for the new characters, then overwrite the string with the new characters”. Any strings overlapping with the string being changed will see corresponding changes in their start, length, and contents.

Here's an example (and please forgive the forward references to *matching* and *altering*; we will discuss those later). We start with a string in **:_dw:** We will use vertical bar characters '|' to indicate the contents of an overlapped string- that is, the start and end (inclusive) of a typical overlapped CRM114 variable (there are non-overlapped variables too- they are called *isolated variables*. As you might guess, they are created by the ISOLATE statement)

The starting string in **:_dw:** is 'All mimsy were the borogroves':

```
:_dw:            All mimsy were the borogroves
```

Now, we can use a MATCH to set an overlapped variable **:a:** on 'mimsy':

```
:_dw:            All mimsy were the borogroves
:a:                  |||||
```

and use another MATCH to set another overlapped variable **:b:** on 'boro':

```
:_dw:            All mimsy were the borogroves
:a:                  |||||
:b:                                 ||||
```

Now, we ALTER the **:a:** from 'mimsy' to 'frumpy':

```
:_dw:            All frumpy were the borogroves
:a:                  ||||||
:b:                                  ||||
```

Notice that both **:a:** and **:_dw:** changed length, because 'mimsy' is not the same length as 'frumpy'. Notice also that **:b:** didn't change length and still has the value 'boro', but it *did* change starting position. CRM114 takes care of all of this start/length bookkeeping automatically, you don't need to consider it in your programs.

Now we ALTER the **:b:** variable from '**boro**' to '**mango** ' (note the trailing space we added to the mango string) yielding:

```
:_dw:            All frumpy were the mango groves
:a:                  ||||||
:b:                                  ||||||
```

Note that **:a:** didn't change, **:b:** changed from '**boro**' to **'mango** ', and **:_dw:** both changed and got longer. All the bookkeeping to take care of this is automatic; you never need to worry about it.

This ability to deal with overlapping strings is one of the unique powers of CRM114. Of course, sometimes you don't want overlapping strings- CRM114 provides a way to make a nonoverlapping copy of a string, which can then be altered without affecting the original. These are called ISOLATEd variables and will be described in detail later; for now, it's enough to know that it's easy to use the ISOLATE statement to create copies of strings that are guaranteed to not be overlapped (and equally easy, given a nonoverlapped string, to use the MATCH statement to superimpose new overlapped variables onto the base string).

CRM114 keeps track of all of the bookkeeping, as to where each variable starts and ends, and what variable or variables may currently contain any particular part of any particular string. Similarly, when a section of text becomes unreferenced, CRM114 can automatically reclaim the string space “in flight”, so there’s never a big pause for CRM114 to do a “garbage collection” because the string space is actively managed continuously.

Variable Restrictions

In some cases, it's useful to restrict the part of a variable we want to access. There are two ways to do this:

1) Match another variable onto the first variable, matching only the part we want to access, and then use that new variable. This is the preferred way to use the part of a variable that is described by a pattern-matching regex.

2) Use a *variable restriction*, that is, restrict what part of the variable we want to access. This is denoted by adding information in **`[box type delimiters]`**. Almost all CRM114 statements that can accept a variable by name can accept a variable restriction, (note – **`:_dw:`** is a variable by name and so can be restricted, but a var-expanded variable like **`:*:_dw:`** is *not* a variable by name, and so can't be restricted.).

Here's an example of a variable restriction:

```
        [ :_dw: 30 10 ]
```

This means “In whatever statement this var-restriction appears, the domain of action is the default data window (that is, **`:_dw:`**), starting at the 30th character (zero-based counting, so character zero is the first character in the string) and including a total of 10 characters. Similarly, you can restrict the part of a variable to that subsection that matches a regular expression (we'll talk about regular expressions in depth later.)

Variable restrictions will be examined in much greater depth later; for now, it's enough to know they exist and can redirect the “subject” of a statement to any variable, or any part of any variable.

Variable Expansion

When a variable name appears in a program, the variable name is just text unless it is marked for expansion. If you want to get at the value of a variable, you must specify that you want the variable expanded. Here's a brief list of variable expansions, and the order of evaluation of expansions; don't worry if these don't make sense yet; they will all be explained in the next few pages.

1. Backslash **`\`** expansions are done first.
2. Variable expansion **`:*:var:`** expansions are done next.
3. Variable indirection **`:+:var:`** expansions are done next.
4. String length **`:#:var:`** expansions are done next
5. Mathematical **`:@:var:`** expansions are done last.

The reason for choosing the above order of precedence, and for performing all expansions of one type before doing any expansions of the next type is simply that the most commonly used actions are usually needed in this order. For example, it's much more common to need the **`:#`** length of a string as an input to a **`:@`** mathematical expression than to need the length of the result of a mathematical operation (especially since the latter is controlled more by the output formatting than by the actual mathematical value). Similarly, a **`:*`** var-expansion is often used as a component of a **`:+`** var-indirection when variable names are being created dynamically in a program, while using a var-expansion on a previously var-indirected value is not particularly meaningful except in contrived situations.

Backslash expansions

First, all \-expansions are done. Note that backslash expansions are *not* real variables, these are actually backslash-escapes of constants, just as in C, and are treated no differently than any literal constant or character.

`\n`	- a newline.
`\a`	- an “alert” (rings the bell. Sometimes known as ctrl-G)

`\b`	- a backspace
`\t`	- horizontal tab
`\v`	- vertical tab
`\f`	- form feed
`\r`	- carriage return (stays on the same line, but returns to column 1)
`\\`	- a backslash
`\/`	- a slash
`\xHH`	- a hex character constant. It must have exactly two hex digits.
`\oOOO`	- a octal character constant. It must have exactly three octal digits.

If you wanted to include something that *looks* like the above – that is a literal **`\n`** in the output, not a newline, you can escape the **`\`** with another **`\`** – that is, **`\\n`** will print as `\n`, (the text, not a newline) and **`\\x4A`** will print as `\x4A.` (In case you are interested, `\x4A` is a capital 'J').

Variable expansion

After the \-codes have been completed, **`:*`** var-expansion is done:

`:*:`*`var`*`:` - expand the ***`:var:`*** to its string value once. This isn't recursive; if the result contains an expansion like **`\n`** that wasn't present in the original string, it will NOT be expanded.

Note that some of the ASCII \-codes are available both as a \-code and as a **`:*:var:`** built-in variable. That's intentional – consider, how do you print out a literal string containing a colon or an asterisk or a slash if colon and asterisk trigger var-expansion, and slash defines end-of-string? By having both **`\`**-code and **`:*:var:`** versions of these characters, we can decide where in the var-expansion pipeline these variables expand.

Variable Indirection

After the entire string has been evaluated for **`:*`** var expansions, the string is then evaluated again for **`:+`** variable indirection operators:

:+:var: - expand the var to a string value twice. This is how you can pass subroutines a variable "by name". If the name of a variable is passed to a subroutine and then **:*** is used in the subroutine on the aliased variable name, the result is the name of the caller's variable, not the value. To get the value the caller had, use **:+** instead. To write the caller's variable, use **:*** to get the name used by the caller, not the value.

Variable indirection is one of the more powerful bits of CRM114 programming. In general, you won't need it unless you start writing programs that use subroutines, so don't fret if the necessity or nuances are unclear at this point. Variable indirection also allows you to create and access new variables "on the fly" which can be very handy in some situations.

Normal var-expansion stops at this point, without making any more passes through the string, and the resulting string is what's used. The result is that it's not hard to have a CRM114 program write CRM114 programs; the **:*:var:** strings that get written are not re-evaluated before printing, because the var-expansion only does these once.

If you use recursive var-expansion (EVAL uses this, as well as the internal parts of **[]**-variable restriction code, which we'll cover below), the string is recursively evaluated, until no more var-expansions are done. In addition to ****-codes and **:*:var:** expansions, two additional expansions are enabled: string length expansions with **:#** and math evaluations with **:@** .

String length and Math Evaluations

The final two expansions available are string length evaluation and mathematical expression evaluation.

:#:*var*: - the length of what **:*:var:** expands to, expressed as an integer string. It is almost always preferable to use the form **:#:var:** (no **:*** used) instead of wrapping a **:#** length measurement around a **:*:var:** expansion, as in the verbose **:#::*:some_variable::** because it is much faster for the **:#:**-code to take the length of the string directly from CRM114's variable tables than to build the string into a buffer, measure the length of the string, and then throw the string away.

After all string length evaluations are completed, the string is re-evaluated one last time for mathematical expressions:

:@:*math_expr*: - the result of doing the math in **:*:math_expr:**, expressed as a decimal string. This evaluator supports both RPN (Reverse Polish Notation) and non-precedence parenthetical algebraic notation, as well as supporting inequality comparisons.

Not every evaluation situation will want all of the evaluations possible; normally only \-codes, **:*** var-expansions, and **:+** var-indirections are done, and they are done as a single pass (so recursion is not evaluated and things that evaluate to embedded \-codes and **:*** or **:+** var-expansions will not actually evaluate those expansions).

If you want **:#:** and **:@:** evaluations, or a recursive evaluation (that is, keep reevaluating the string until it stops changing), you will usually need to use the extended evaluator (see the EVAL statement for this). EVAL uses hash-based heuristics to detect with very high probability when a recursive evaluation will infinitely loop – this detector can be fooled (CRM114 does NOT have a perfect solution for the Halting Problem!) but it's handy to have your program tell you what is probably wrong rather than wedging in an infinite loop. The default loop depth maximum is currently 4096 attempts; if the number of iterations exceeds that (or a loop is detected before that) then recursive evaluation is aborted and a user-TRAPpable FAULT is triggered.

Mathematical strings

The **:@:mathematical_string:** evaluation takes the result of the mathematical string (which may include :*:var: expansions and :#:var: length expansions) and does the math. The incoming string should be an algebraic or RPN (Reverse Polish Notation) expression, expressed as human-readable text (base-10 numbers are assumed; there is no express provision for other bases[7]). CRM114 will do all of the necessary conversions from the human-readable format to double-precision IEEE floating point, and then perform the specified mathematical operations. The math itself is done as IEEE double-precision floating point, and the result is converted back to a human-readable string and inserted into the string replacing the :@:mathematical_string: text.

You might wonder why a string-oriented language like CRM114 doesn't do infinite-precision string based math like bc does, instead of IEEE double-precision floating point. The wisecrack answer is "if you want bc-style math, use bc[8]!" The real answer is that IEEE floating math is considerably faster (because modern CPUs have hardware floating point

7 Some versions of CRM114 may have support for hexadecimal input and output. It also depends on what the local C library's implementation of **strtod()** is willing to accept.

8 bc is a common open-source console-based infinite-precision arithmetic calculator application.

units, while string math is still interpreted one character at a time) and IEEE floating point is perfectly adequate for most CRM114 uses (as most math is used to manipulate classifier probabilities, which are approximate numbers anyway).

Default Format of Mathematical String Calculated Values

The following procedure is used to format the result of a **:@** math operation (which is *not* padded with spaces on either end) if no format specifier is included in the mathematical string:

1. if equal to floating-point 0, then as “0”, otherwise
2. if an integer less than 10^{12}, print as an integer (no decimal point), otherwise
3. if greater than 10^{12}, print in E-notation, 5 decimal digits after the decimal, otherwise
4. if within 0.01 of zero, print in E-notation, again with 5 digits after the decimal, otherwise
5. print as a decimal number, with five digits after the decimal.

This procedure is a compromise between long-form mathematical accuracy and easy human readability. If it is necessary to change the procedure, an output-format operator can be put into the string to be evaluated. If you need something really bizarre, the CRM114 output routine can be changed; it is easily done with alteration to the source code (at this writing, in `crm_math_exec.c`).

Math Evaluation Operations

CRM114 math evaluation supports both RPN (Reverse Polish Notation) and non-precedence algebraic notation (algebraic notation that is evaluated strictly left-to-right except where parentheses override).

The following items are supported inside the strings in both RPN and algebraic **:@** expressions

- The leftmost character may optionally be an **A** or an **R** . If either of these is present, it forces the evaluation to be in **A**lgebraic or **R**PN notation. If neither is present, the default is taken from the **-q** flag, with the default default being algebraic notation. This only works as the first character; you cannot start out in one notation form and switch to the other in mid-expression.
- numbers – may be expressed as integers (no decimal point), decimal numbers (with a decimal point, optionally with some digits after the decimal point), floating-point in E-

notation, and hexadecimal integers. Fortunately, these formats are compatible with most other computer languages.[9]

  - No spaces are allowed within a number, so negative numbers *must* have the leading minus sign directly adjacent to the most significant digit.
  - If the number is written in E-notation, there must be no spaces between the mantissa and the E, nor between the E and the exponent.
  - If the E-notation exponent is negative, then there must be no spaces between the E and the minus sign in the exponent, nor between the minus sign and the digits of the exponent.
  - both upper and lower case E is acceptable.
  - the IEEE "special" numbers such as NAN (Not A Number) and INF (infinity, both positive and negative) are supported (and are case independent).
  - Hexadecimal number inputs must be prefixed with **`0x`**, such as **`0xDEADBEEF`** .

- At least one space must separate each number and each operator from any other number or operator.
  - A "space" can be a blank space, a tab, or a newline, so "cash register tapes" that separate numbers on new lines are valid inputs for math evaluation.

- The following operators are allowed. There is no operator precedence in algebraic mode- operators are evaluated left-to-right as encountered unless parentheses are used to force another grouping. The operators are:

+	addition
-	subtraction (dyadic – if you want monadic negation, subtract from zero)
*	multiplication
/	division
%	modulo
>	greater-than (0/1 valued)
<	less-than (0/1 valued)
=	equal-to (0/1 valued)

- Parenthesis, **`(`** and **`)`** , are allowed only in algebraic mode, not in RPN mode.
- Output formatting operators – **`e, E, f, F, g, G, x, X`** only. These yield the same results as they do in a ANSI C `sprintf`[10]. For **`e, E, f, F, g,`** and **`G`** formats, the formatting occurs immediately upon operation execution (so, if you want to round to

9 These rules happen to be exactly compatible with ANSI C's `atof()` standard-- because CRM114 uses ANSI C's atof`()` routines.

10 Unsurprisingly, the formatting operators behave exactly as the ANSI C sprintf function does, because CRM114 uses **`sprintf()`**. To prevent possible attacks, <u>only</u> the operators above are allowed, and no back-counts or other extended formatting is permitted.

three decimals part-way through an expression, you can). Additionally, the last formatting operation encountered is re-applied to the final string's mathematical value for output. The hexadecimal format operators **x** and **X** do not cause loss of precision; they only set the final format of the output (which is first coerced to a 32-bit integer and then output)

- In algebraic mode, the formatting operation specified by the operator is applied to the left operand; the field width specification (including decimal width, such as the **12.6** in **12.6f**) is supplied by the right operator.
- In RPN mode, the formatting operation is specified by the operator, the top-of-stack specifies the field width (including decimal width, such as 12.6F) , and the numeric value is supplied by the stack value one below the top of stack.
- Hexadecimal format operators **x** and **X** do not prefix their output with **0x** so to output the number in a re-readable fashion, you must supply the **0x** externally (this is in compliance with the ANSI C standard for hex output, and incidentally is also what Intel .HEX files need for programming embedded microcontrollers) .

- No other characters (including alphabetic characters) are allowed; if an unrecognized alphabetic character appears in the mathematical string, a warning will be triggered.
  - Note that because a **:*:var:** evaluation will happen before the **:@:** evaluation happens, any **:*:var:** that evaluates to a number is OK.

If the expression used one of the relational operators >, < or =, that logical result will be used to alter program control (see the EVAL statement for more details. The relational operators >, < and = also yield a 0/1 result, so it is easy to add up the number of logical conditions satisfied.

Here's an example of math evaluation, We apologize for using the EVAL statement; we'll explain it in depth later. For now, please accept that it evaluates its **/pattern/** repeatedly until it ceases to change, and then alters the **(:var:)** to that value. We'll put the output in the default place, that is, **:_dw:** .

We will evaluate (2 + 3) * (8 * –2.5) as a algebraic-notation string. Evaluating left-to-right (and overriding only when there are parentheses) works out as 2+3 is 5, then 8 * -2.5 is -20, so the result should be 5 * -20 = -100.

Here's the code:

```
{
        eval (:_dw:) / :@: ( 2 + 3 ) * ( 8 * -2.5 ) : /
        output /:*:_dw: \n/
}
```

When you run this program, you get:

```
# crm mathdemo1.crm
return, then ctrl-D
 -100
#
```

which is the correct result.

We can make a simple calculator by using the contents of **:_dw:** as our input to the evaluation. Here's the code:

```
{
      eval (:_dw:) / :@: :*:_dw: : /
      output /:*:_dw: \n/
}
```

Notice how we are nesting **:*:_dw:** *inside* the **:@:*var*:** math evaluation. That works fine, because (remember from above) that \-constants evaluate first, then **:*** var-expansions evaluate, then **:+** var-indirections, then **:#** length expansions, and finally **:@** math evaluations. Here, the expression `:*:_dw:` gets substituted by whatever we typed on `stdin`, and then <u>that</u> result gets math-evaluated.

When we run this code, we get:

```
# crm mathdemo2.crm
1 + 2 + 3 + 4
return, then ctrl-D
 10
#
```

No surprises there. Let's try it with some floating point numbers:

```
# crm mathdemo2.crm
1 + 10 + 3.14
return, then ctrl-D
 14.14000
#
```

Since the result was a floating-point value, CRM114 shifted into a decimal output representation. If we wanted to specify an output format (say, the equivalent of `%15.9E` in C's `printf()` output), we can do that right in the mathematical string we are evaluating:

```
# crm mathdemo2.crm
1 + 2 + 3 + 10 + 3.141592865721
 19.14159
# crm mathdemo2.crm
1 + 2 + 3 + 10 + 3.141592865721 E 15.9
 1.914159287E+01
#
```

You can use formatting to round off an intermediate value in a calculation, then apply a different format to the final value. Here's an example – note how we rounded off pi to 3.14 (with the **f 4.2**) before the addition, then reformatted the result to four digits (with the **f 6.4**) after the decimal:

```
# crm mathdemo2.crm
( ( 1 + 3.14159265385 ) f 4.2 ) - 1 f 6.4
 3.1400
#
```

We can use the RPN mode of CRM114 to make a nice little RPN calculator; we do this by specifying **-q 1** on the command line (this command flag, as well as others, will be covered in the next chapter). Here, we'll calculate the area of a 12 cm diameter circle with the formula A = πr^2

```
# crm mathdemo2.crm -q 1
6
6 *
3.14 *
return, then ctrl-D
 59.15760
#
```

So, the area of a 12 cm circle is about sixty square centimeters, and we also see that we can use newlines to separate values and operators in a math expansion.

Hexadecimal input and output happens with equal ease; here's an example showing the input of hex 0x4000 to decimal:

```
# crm mathdemo2.crm
0x4000
 16384
#
```

confirming that 16K (decimal) is 4000 hex. Conversion from decimal to hex is equally painless (note that hex output with **x** and **X** do not include a **0x** prefix; if you want that, you must include it yourself; this is the same as in ANSI C)

```
 # crm mathdemo2.crm
1000000 x 8.0
    f4240
#
```

For more information on math evaluation and inequality testing, see the EVAL statement.

Command Line Flags and Variable

Not only do CRM114 programs default to read standard input to EOF before execution, but it is possible to configure the CRM114 engine from the command line, as well as conveniently set variables that the running program will "wake up" seeing.
When CRM114 is invoked, the command line is parsed before the program source code is loaded and the first pass of compilation started.

Command-line Flags

The following flags will control various aspects of CRM114 execution. They may be placed in any order. Don't worry if some of these flags don't make sense yet. They will eventually, and simply knowing that these flags exist will be a help to you.

`-d N`	-run N cycles, then drop into the CRM114 debugger. If no N, debug immediately after reading standard input to EOF.
`-e`	- no environment variables imported. By default, all of your shell environment variables are imported and assigned as non-overlapping (ISOLATEd) variables in the system namespace of variables that start with **`:_`** . For example, the shell environment variable SHELL (which, for most Linux users, is `bash`) will be imported as the ISOLATEd variable `:_SHELL:` with the value "`/usr/bin/bash`".
`-E N`	- set exit code starting base value. normally the CRM114 engine itself will use only two exit code values, those being the OS-defined EXIT_SUCCESS and EXIT_FAILURE code values, and all other exit values are available for the user program. In some cases, a much more detailed exit analysis is desired; in that case set the exit code base value **`N`** to a positive value greater than 1 and greater than the largest exit code the user program will use. This sets the CRM114 engine to use exit values starting at **`N`** plus one. In this mode, every exit in the CRM114 system has it's own unique code value. This is recommended for system developers only.
`-h`	- print help text, including a short list of flags.
`-l N`	- print out a prettyprinted program listing. N defines the detail level, from

0 no listing printed at all (the default)

1 simple prettyprint, with nice indenting, suitable for documentation or re-execution

2 add line numbers

3 add nesting levels

4 show per-statement FAIL, LIAF, and TRAP vectors

5 show the JIT pre-parse information

-p - generate an execution-time-spent profile on exit. This profile will show the cumulative amount of time spent in each line of your program. The granularity and unit size of this counter is the clock tick time of your system. Lines of code with less than one granularity of time spent will not be printed, so as to not clutter the output.

-P N - maximum number of program lines. Part of the anti-DoS profile of CRM114 is that program size post-preprocessing is limited; the default limit is 10,000 lines; this should be adequate for most users.

-q M - what mode to run mathematics in. **M** can have the values 0, 1, 2, or 3. Mathematical expressions can override this setting on a per-expression basis with the **A** or **R** expression operators.

0 (zero) is the default, and does mathematics expansions only in EVAL statements in non-precedence algebraic mode.

1 is mathematics expansions only in EVAL statements, and in RPN notation.

2 enables mathematics expansions in all var-expansions, in non-precedence algebraic notation.

3 enables mathematics expansions everywhere in RPN notation.

-s N - new feature file (.css) size for all classifiers is N feature slots (see the LEARN and CLASSIFY statements for details on feature slots). The default 1 million +1 feature slots should be sufficient to get most users started.

-S N - new feature file (.css) size for all classifiers is N rounded up to 2^N+1 feature slots. There is a small performance advantage to feature slot counts on a 2^N+1 boundary.

-t - enable user trace output, with line numbers matching the pretty-printed annotated program listing. This level of tracing is useful for a typical user trying to figure out what went wrong with their program.

-T - enable implementor's trace output. This level of tracing is very deep, and provides diagnostics on the detailed actions of the execution engine. It is very verbose, and should be considered only for system implementors and the truly masochistic.

-u *dir* - chdir to directory *dir* before loading the program and starting execution; all filesystem references will be relative to the *dir* specified. This includes the file containing your source code; it must either be an absolute file name (starting at '/') or it must exist in the changed-to *dir* or a subdirectory of *dir*.

-v - print the CRM114 version identification string (which includes the regex library identification string) and then exit. Please include this string on all bug reports; it helps immensely to know what exact version and release of the software exhibits the buggy behavior.

-w N - maximum size of the data window (in bytes). This is the absolute limit on the size of the data window; the default is 8 megabytes. Like the limit on the maximum number of program lines, this is part of CRM114's anti-DoS profile.

Command-line Variables

The command line can also contain variables that will be set in the CRM114 execution environment before execution begins.

These command-line variables are set as follows:

-- – (two minus signs) signals the end CRM114 flags; prior flags are not seen by the user program; subsequent flags are not used by CRM114 initialization but are visible to the user program. This is very handy if you need to write a program that needs to use a -flag that is already in use by CRM114 itself. Putting '**--**' before the local-use program flags will keep CRM114 from using your program's flags, and also keep your program from using flags meant for CRM114.

--foo (two minus signs) - creates the ISOLATEd user variable ***:foo:*** with and initializes it with the value 'SET'. Your program can easily test to see if the **–foo** option was supplied because it will have the value 'SET'. (Later on, when you learn about the ISOLATE statement, you will see how to set a default value for a command line variable as well.)

--x=y - creates the ISOLATEd user variable ***:x:*** with the value **y**

-{ program statements } - execute the statements inside the { } brackets.

Absent the **-{ program statements }** flag, the first arg is taken to be the name of a file containing a CRM114 program, subsequent args are merely supplied as **:_argN:** values. It is wise to always use single quotes around command line programs `'-{ like this }'` to prevent the shell from doing odd things to your command-line programs.

Almost all OS kernels respect the “magic number” criterion that if the first two bytes of a file are **#!** then the first line of the file contains the ASCII name of the execution engine that is capable of running the file. CRM114 programs can use this; any of your programs can be invoked by the shell if the file protections are set to indicate “executable file” and the first line of your program file uses the #! standard, as in:

```
#! /usr/bin/crm
```

Some kernels allow extra information to be encoded on the #! line, other versions don't (for example, Linux does, and BSD and Solaris don't). Therefore, in Linux (and only Linux), a program can specify what the options it can accept are- and what values those options can take. CRM114 will add this information to the output of the **-h** (help) flag output. Unfortunately, because not all kernels support this, it's not a portable feature.

To use this feature, add a parenthesized list of the parameter names (and values) that you want your program to accept. For example:

```
#!/usr/bin/crm  -( var1 var2=A var2=B var2=C )
```

means that your program will accept **--var1** with any value, and **--var2** only with the values of **A**, **B**, or **C**. Here's an example of a command line passing in program args and values. (For now, accept that the ISOLATE is a no-op if the variable is already ISOLATEd, and will create an empty variable if it doesn't already exist as an ISOLATEd value).

Note that we can invoke this program directly from the command line (`bash` reads the **#!** and invokes CRM114 for us) :

```
#!  /usr/bin/crm  -( var1 var2=A var2=B var2=C )
isolate (:var1: :var2:)
output / Var1 = ':*:var1:', Var2 = ':*:var2:' \n/
```

When we run this program, we get no surprises about **`:var1:`**

```
# ./cmdline_1.crm
return, then ctrl-D
 Var1 = '', Var2 = ''
# ./cmdline_1.crm --var1
return, then ctrl-D
 Var1 = 'SET', Var2 = ''
# ./cmdline_1.crm --var1=Hello
return, then ctrl-D
 Var1 = 'Hello', Var2 = ''
```

As expected, **`:var1:`** can be set as desired, from the command line.
When we try using **`:var2:`**, we find something more interesting.

```
# ./cmdline_1.crm --var2=A
return, then ctrl-D
 Var1 = '', Var2 = 'A'
# ./cmdline_1.crm --var1 --var2=A
 return, then ctrl-D
Var1 = 'SET', Var2 = 'A'
# ./cmdline_1.crm --var1 --var2=C
return, then ctrl-D
 Var1 = 'SET', Var2 = 'C'
# ./cmdline_1.crm --var1=Hello --var2=C
return, then ctrl-D
 Var1 = 'Hello', Var2 = 'C'
```

Because we listed the set of possible values that **`:var2:`** can have, CRM114 tells us that we are doing something wrong if we go outside that list, to wit:

```
# ./cmdline_1.crm --var1=Hello --var2=D
 ***Warning*** This program does not accept the flag ' var2=D ' ,
 so we'll just ignore it for now.
return, then ctrl-D
 Var1 = 'Hello', Var2 = ''
# ./cmdline_1.crm --var3
 ***Warning*** This program does not accept the flag ' var3 ' ,
 so we'll just ignore it for now.
 return, then ctrl-D
Var1 = '', Var2 = ''
```

```
      #
```

CRM114 also “knows” what's legal for this program- if you ask this program for help, you'll get a hint as to what the valid flags are for this program:

```
      # ./cmdline_1.crm -h
      [... flag information omitted ...]
       This program also claims to accept these command line args:
        ( var1 var2=A var2=B var2=C )
      #
```

If you want to guard your program against being given any other flags or flag values than your want it to have, add a '**--**' (two minus signs) after the parameter list. Since **--** will block any further command line flags, this will also block **-h** for help; your program will need to test for a **-h** help-request flag in the **:_argN:** variables and provide its own help system.

REGEXes In Depth

This chapter is the full explanation of regular expressions – *regexes*. This chapter deals mainly with how to *use* regular expressions; they're rampant in CRM114, so knowing how to use a regex is more or less mandatory for CRM114 programming. There is also a deep mathematical theory of regular expressions, (which we won't get into) and a good body of work concerned with how to efficiently implement regular expressions (which we will only touch on briefly, when it impacts us).

Regexes are an extension of *wildcarding* that you've probably used in your computer work, where you typed

```
        ls *.txt
```

to get a list of all of the .txt files in a directory. Regexes expand on the concept of wildcarding to provide much finer granularity – and to capture submatches within the matched text.

The concept of a regular expression is intimately connected with the concept of a *recognizer*, which you might ... um... recognize if you have a background in computer science. The concept of a regex is also connected to the concept of BNF syntaxes, but BNF syntax spreads the acceptable options over pages of productions, while a regex will typically be self-contained.

Think of a regex as a very compact finite-state machine that looks for any of a (possibly infinite) set of possible acceptable inputs, and returns just one bit: “ matched” or “did not match”.

NB: A regex by itself usually doesn't start with a `/` and end with a `/`; the slashes you see in CRM114 programs are the CRM114 declension, not part of the regex itself. Therefore, we will **not** be enclosing standalone regex patterns in `/` characters `/`like this`/` in this chapter.

Regex Libraries

A regex string is a compact representation of a generalized pattern to match. The program that actually performs the matching, using the regex as data, is called the _regex library._ and you can use the one built into your copy of CRM114.

Not all regex libraries are the same. There isn't even a single standard specification for a regex library. There are several similar-but-different specifications, such as the POSIX regex specification, the PERL specification, etc. The GNU regex library (based on the POSIX specification) is included in most *NIX systems and is also available for most other computing platforms. Although CRM114 can use the GNU regex library, we recommend instead to use the TRE regex library instead.

The TRE regex library, also based on the POSIX specification and also licensed under the GPL, is available in source form and runs on most platforms (the source is included in the extended CRM114 kits). TRE is the default CRM114 regex library because the TRE library has more features and fewer bugs. More specifically:

- TRE is NULL-safe (that is, it can search strings that contain embedded `\x00` characters, which GNU regex cannot deal with. To GNU regex, a NULL always means "end of string", and there's nothing that the CRM114 matching engine can do to get around that. To be honest, the bug there isn't in GNU regex- the bug is in the POSIX API which specifies a null-termination instead of a counted-length string).
- TRE has an approximate matching capability, quite handy for filtering.
- TRE supports the **`?-`**extensions for substring case independence, substring newline bridging, and substring associativity.
- TRE supports the **`\Q \E`** literal-text mechanism
- TRE is under active maintenance, support, and extension
- TRE is considerably faster than GNU regex, and runs in constant memory once compilation of the regex has completed.
- TRE has support for wchars (wide characters, such as are useful to support languages like Japanese and Chinese, although this support is not used in CRM114).

In this chapter (and this chapter _only_) we will discuss the differences between the GNU and TRE regex libraries; in all other chapters we will assume you are following our advice and using the TRE regex library.

> If for some reason you want to use yet another regex library, you will only need to rewrite five functions in the CRM114 runtime system – these are the functions to compile a regex, to execute a regex, to release memory allocated during regex compilation, to get the string representation of a regex error, and to get the string representation of the regex library's name and version.

Regex Characters

Most regex libraries conform to the POSIX standard of "first match, then longest fit". That is, when there are multiple places where a match can be made in a string, the first character where the match can start is always used, then the longest possible match from that starting point is returned.

Most characters in a regex match themselves – for example, **a** through **z** and **A** through **Z** match themselves. We've seen this above, in the `hello`-finding examples. An **a** will match only an **a**, not an **A** and not a **b**, and it will match only one **a**.

Besides the letters a through z and A through Z, the numbers **0** through **9** usually match themselves (but see below- numbers have other uses too.)

Punctuation marks are not so simple. Some punctuation marks have a special meaning in a regex. For example, a **.** (a period) means *match one character's worth of any character*. The characters * (asterisk), **+** (plus sign) and **?** (question mark) specify repetition. **{}**, **[]**, ^, **$**, and several other characters also have special meanings. To actually represent any of these characters as themselves in a regex, you must *escape* them with a \ (a backslash), so a "**.**" (a real period) becomes **\\.** in the regex.

Special Characters in a Regex

Here is a list of regex special characters supported by most regex libraries (though there is some variance), and the special character meaning. This list is not exclusive – Appendix A contains a more detailed listing including the TRE extensions:

. - the 'period' char, matches any single character

***** - asterisk, meaning repeat preceding 0 or more times

+ - plus, meaning repeat preceding 1 or more times

? - question mark repeat preceding 0 or 1 time

***?, +?, ??** - same as preceding, but shortest match that fits, given the already-selected start point of the regex. (only supported by TRE regex, not GNU regex). This is the opposite of the POSIX standard of "first match then longest string".

[ ] - any one of the characters between the **[ ]** characters; when used with

- any span of characters in ASCII sequence, and with **^** as the first character, any character *not* in that set. Except for **]**, **-**, and **^**, all other characters inside the **[**list**]** have no special meaning. Here are some examples:

[abcde] matches any one of the letters a, b, c, d, or e

[a-q] matches any one of the letters a through q (just one of them)

[a-eh-mqzt] matches any one of the letters abcdehijklmqzt

[^xyz] any one character EXCEPT one of x, y, or z

[^a-e] any one character EXCEPT one of a through e

To use a **–** in a character list, make it the first character; likewise to put a **]** in the list, make it the first character. Finally, to have a **^** in the list, make it any character *except* the first character.

{n,m} -repetition count: match the preceding at least n and no more than m times (sadly, POSIX restricts this to a maximum of 255 repeats. Nested repeats like this monstrosity **(.{255}){10}** will work, but are very very slow). Using a var restriction is a good alternative to nested repeats, if the count is fixed.

^ -as first character of a match, matches the start of a line (ONLY in <nomultiline> matches; otherwise this matches only at the 'start of input variable')

$ -as last character of a match, matches at the end of a line (ONLY in <nomultiline> matches, otherwise this matches only at the 'end of input variable')

(a parenthesized regex string) - the () go away, and the string that matched inside is available for capturing. Use **\(** and **\)** to match actual parentheses. This is how a subexpression can be connected to a variable. This is also how you can specify a group of characters to all be affected by something like a repeat operation.

foo|bar -match "foo" or "bar" (the OR bridges the "entire left side" and the "entire right side", so **foo|bar** matches **foo** or **bar**, not **fooar** or **fobar**). To get a shorter extent of ORing, use parentheses, e.g. the regex **f(oo|ba)r** matches either **foor** or **fbar**, but not **foo** or **bar**.)

The following are POSIX expressions that represent various useful ranges of characters. These mostly do what you'd guess they'd do from their names.

`[[:alnum:]]`	alphanumerics (a-z, A-Z, and 0-9)
`[[:alpha:]]`	alphabetic (a-z and A-Z)
`[[:blank:]]`	space and tab only
`[[:space:]]`	"whitespace" (space, tab, vertical tab (^K), \n, \r, ..)
`[[:cntrl:]]`	control characters (ASCII values 0x00 through 0x1F)
`[[:digit:]]`	numerics (0 through 9)
`[[:lower:]]`	lower-case characters (a through z)
`[[:upper:]]`	upper-case characters (A through Z)
`[[:graph:]]`	any character that puts ink on paper or lights a pixel
`[[:print:]]`	any character that moves the "print head" or cursor.
`[[:punct:]]`	punctuation characters
`[[:xdigit:]]`	hexadecimal digit (0-9 and a-f)

TRE also supports the following special expressions for looking backward in the text previously matched, for verbatim matches (matches with no special characters whatsoever), non-capturing parenthesized expressions, and case independence, nomultiline, and right-associative matching:

`\1` through **`\9`** - (TRE only) matches the Nth parenthesized subexpression. You don't have to backslash-escape the backslash (e.g. write this as **`\1`** or as **`\\1`**, either will work)

`\Q` - (TRE only) start verbatim quoting - all following characters represent exactly themselves; no repeat counts or wild cards apply. This is *only* terminated by a **`\E`** or the end of the regex.

`\E` - (TRE only) end of **`\Q`** verbatim quoting.

`(?:some-regex)` – (TRE only) parenthesize a subexpression, but don't associate an overlapped variable with the subexpression. This is handy when you have four or five overlapped variables in a match, and need to insert a parenthesized match string that you don't want associated with a variable.

`(?inr-inr:regex)` – (TRE only) let you turn on or off case independence, nomultiline, and right-associative (rather than the default left-associative) matching. These nest as well.

As you can see, the extra features of TRE regex over GNU regex make a strong case to use TRE regex. Additionally, TRE can deal with strings containing embedded NULLs, another strong reason to use it.

With that, we'll assume the use of TRE regex for the rest of this document.

Section 3:
CRM114 Statements In Detail

This section discusses each of the CRM114 statements in detail, including many short examples.
As we've discussed before, there's really no way to give examples of CRM114's statement usage without making forward references to statements that themselves haven't been described yet. For this, the author apologizes; there are just some statements whose meaning you'll have to accept on faith for now.

To whet your appetite, here's the summary for the next section of the book:

- MATCHing – a combination of searching, assignment, and branching
- ALIUS – a combination of ELSE and CASE
- LIAF, FAIL, GOTO, and Statement Labels – the basics of program control
- INPUT and OUTPUT – basic I/O
- ACCEPT and WINDOW – advanced I/O
- ALTER, EVAL, and HASH – surgical string alterations
- ISOLATE – non-overlapped string creation
- CALL, RETURN, and EXIT – advanced program control
- SYSCALL – dynamically scoped control, forking, and calling other programs directly
- INSERT – inserting code from other files
- UNION and INTERSECT – string-set operations
- LEARN and CLASSIFY – using the built-in text classifiers
- TRAP and FAULT – error catching and fix-up

The MATCH statement

The MATCH statement is one of the most common statements in CRM114. With MATCH, one can search for strings inside other strings (including regex searches, which as you saw above, are quite powerful). MATCH also can set an overlapping variable (so, MATCH is an assignment statement). Lastly, MATCH can change the program flow so MATCH is like an IF-statement.

The syntax of MATCH is a bit complicated; don't let this throw you. We promise that it will all become clearer soon (remember, except for the word MATCH at the start of the statement and having a regex pattern somewhere in the statement, all arguments are optional and may appear in any order).

```
match /regex_pattern_to_match/
      [:optional_var_to_match_in:]
      [:optional_var_to_match_in: start length]
      (optional_vars_to_put_values_on)
      <optional control flags>
```

Breaking this down – MATCH means this is a match statement. Then the regex is a pattern to be matched; these two are both required, and MATCH must come first.

The first optional argument is the variable we want to match in; by default, we do the match in the data window **:_dw:**. If you want, you can use a variable restriction (which we'll shortly explain) to limit the part of the variable available to match in.

Then there are optional "match variables". The first match variable is set to point to the string that the regex matched (as one of CRM114's overlapping variables). Match variables 2 through N are pointed to the first through Nth parenthesized subexpressions in the regex; this is in left-to-right order of the opening parenthesis of the subexpression.

Finally, the optional match control flags allow the programmer to control just how the match is to be done. There are a number of match control flags; we list them all below.

Remember, the MATCH command must go first (it may be capital letters or lowercase) and there must be a regex. Other than that, the arguments may go in any order, and all but the command MATCH and the regex are optional.

A Simple Regex Match Example

Here is a simple example program that uses a MATCH to regex-match on a string, and sets a variable to the matched section. These are the same regexes described the previous chapter, and now we can see them in action.

Here is the program:

```
{
      match (:my_first_variable:) /foo 123 bar/
      output /My first variable -> :*:my_first_variable: \n/
}
```

If you put this into the file "my_first_variable.crm" and run it, you get:

```
# crm my_first_variable.crm
see if match can find this foo 123 bar in this sentence
return, then ctrl-D
My first variable -> foo 123 bar
#
```

and it works as expected – it matched "foo 123 bar". But what happens if we don't have a "foo 123 bar" in the text?

```
# crm my_first_variable.crm
here, we won't have the foo then 123 then the bar,
all next to each other.
What will happen?  All the pieces are here...
return, then ctrl-D
#
```

Nothing happened. Because the regex did not match exactly, (all of the parts of "foo 123 bar" were there, but not adjacent to each other) the regex failed, so the MATCH statement wasn't successful, and so the program never went on to the OUTPUT statement. This "do nothing" is the default for CRM114 – unless the right things are seen, nothing comes through.

Remember this as standard CRM114 behavior:

"When in doubt, nothing out!"

Let's take the program apart. We already know what the OUTPUT statement does – it var-expands its argument, and sends the result to `stdout`. The MATCH statement is:

```
  match (:my_first_variable:) /foo 123 bar/
```

and has the pattern **`/foo 123 bar/.`** This means the MATCH statement will succeed only if it finds the string '`foo 123 bar'` in the input. Since there was no such string, the MATCH statement failed and the program exited without executing the OUTPUT statement.

Now, what if "`foo 123 bar`" is found? Let's try it:

```
      # crm my_first_variable.crm
      Let's see what happens if foo 123 bar is in the input.
      return, then ctrl-D
      My first variable -> foo 123 bar
      #
```

So, the MATCH succeeded in finding "`foo 123 bar`", set the value of **`:my_first_variable:`** to be that string, and execution passed on to the OUTPUT statement. The MATCH not only searched for a strings but it also set a variable value, and also somehow altered the flow of program execution. If a MATCH fails to find its target regex, then the program execution doesn't appear to continue.

To be more exact, here's what MATCH really does:

- The MATCH statement tries to match its **`/regex pattern/`** in the text.
- If the MATCH succeeds, then:
    - any variables in the match are set to the place where the /pattern/ matched
    - program execution continues with the next statement.
- If the MATCH does not succeed then:
    - nothing about the variables in the match are changed
    - the program execution skips downward to the end of the enclosing **`{}`**-block.

So, MATCH searches for strings, sets values, and acts like an IF statement, all at once.

Here's a sample program that shows how this "skip to end of **`{}`**-block" works:

```
      {
            {
                  match /hello/
                  output /Found a "hello"\n/
            }
            output /and now goodbye\n/
```

```
        }
```

This “skip to end of the **{}**-block” is called a FAIL (remember that; you'll hear more about that later).

When we run it, we get:

```
        # crm hellogoodbye.crm
        just give it a hello
        and see what happens
        return, then ctrl-D
        Found a “hello”
        and now goodbye
        #
```

and now we'll run it again, but we will not give it a “hello” in the input:

```
        # crm hellogoodbye.crm
        there is no "h-word" in this input
        so the match should fail.
        return, then ctrl-D
        and now goodbye
        #
```

There was no “hello” in the input, so the MATCH failed and if a MATCH fails, not only do variables not get assigned values, but the rest of that **{}** block doesn't run either.

Attaching Variables in a MATCH

CRM114 supports the concept of attaching variables to the full regex match, and to submatches within a match. This is useful when the data you're manipulating has some sort of internal structure, even if it's just sequential fields in an ASCII text (such as a CSV spreadsheet). These are the parenthesized “captured match variables”; the first match variable covers the entire text that the regex matched, and each subsequent match variable covers the next parenthesized regex subexpression, from left to right. If the regex has nested subexpressions, the resulting match variables will also be nesting. Note that these are overlapping variables, so anything that changes the actual value of any of these variables (rather than capturing a different string with a new MATCH statement) will change the apparent value of all of the variables that overlap the changed area.

This capturing is automatic, so occasionally you may be in the situation where you don't want to capture a variable but the regex syntax doesn't seem to allow you not to capture (for instance, it's automatic that if you have a variable list, the first variable in the list will capture the entire top level matched string). In situations like this, use the the *anonymous variable* named **`::`** (written as two colons, with nothing in between). This is a real variable and can go anywhere you would use a variable, but by common agreement **`::`** is never used for anything except places where the language forces a value that you need to throw away.

Here's an example of a match that grabs first name, last name, and age from an employee flat file. Assume that the input will be one line of the flat file at a time (NB: CRM114 can actually do this one-line-at-a-time splitting for you; see the WINDOW statement for how).

Here is a sample input line of data:

```
      Lincoln Abraham President 12-Feb-1809 Deceased
```

and here's a MATCH statement that would grab the first and last name, and the date of birth . (note that this MATCH statement spans two lines – a backslash like **`\`** as the last character of a line means "continue on next line"):

```
      match (:: :last: :first: :dob:) \
        /([[:alpha:]]+) ([[:alpha:]]+) [[:alpha:]]+ ([[:graph:]]+)/
```

Note that we have four variables in the parentheses, not three. The first variable is the anonymous variable **`::.`** This is the variable to use if you want to throw away that particular result.

Following **`::`**, we have the variables **`:last:`** , **`:first:`** , and **`:dob:`** . These will be assigned to the first three submatches in the pattern. (The submatches are the parenthesized subexpressions in the regex pattern).

The first two submatches are **`([[:alpha:]]+)`**, which matches a string of one or more alphabetic characters. These first two submatches will be attached to the variables **`:last:`** and **`:first:.`** The third submatch is **`[[:alpha:]]+`** *without* parentheses, this matches the "occupation" string, but without parentheses it doesn't create a submatch to attach a variable to (and so the variable doesn't get attached to "President").

The fourth submatch is **`([[:graph:]]+)`**, which matches a string of one or more "graphic characters", that is, any character that when printed actually puts out ink on paper, or lights a pixel on the screen. This will match all letters, all numbers, and all punctuation.

Here's a complete (silly, but complete) program that uses this match statement:

```
      {
          match (:: :last: :first: :dob:) \
        /([[:alpha:]]+) ([[:alpha:]]+) [[:alpha:]]+ ([[:graph:]]+)/
          output /Last= :*:last: \nFirst= :*:first:\nDoB= :*:dob:\n/
      }
```

When run, we get:

```
      # crm submatches.crm
      Lincoln Abraham President 12-Feb-1809 Deceased
      return, then ctrl-D
      Last= Lincoln
      First= Abraham
      DoB= 12-Feb-1809
      #
```

Match Control Flags

The regex specifies the string to be searched for; however the statement itself can affect the context where the search occurs. Most regex-using statements in CRM114 can take a series of *match control flags* that determine how the regex string is to be treated (not all; for some, the behavior of the statement itself overrides the match control flags.

The match control flags also can specify what part of the input should be considered eligible for matching, based upon the most recent successful match within this variable. The CRM114 runtime system keeps track of where the most recent match on a variable was satisfied (start and length); the eligibility of that substring can be controlled with match control flags. The direction of the match can be controlled, as well as the sense of "success" inverted (the MATCH "succeeds" if and only if the regex pattern was NOT found).

Here's a list of the match control flags:

`<absent>` - Inverts the sense of the test; the statement succeeds if the regex does *not* match the string and FAILs if the string is found. Note that **`<absent>`** does not interact with **`<nomultiline>`** to give a "search for the first line that does not contain this pattern", it means "if this pattern cannot be found anywhere in the search area, succeed, otherwise fail". If you need "first line not containing", skip down to Idioms and Tricks: Finding Lines that Don't Contain".

`<nocase>` - ignore case when matching globally over the entire string.

`<literal>` - Treat all characters in this regex as themselves, so none of the characters have any special effects. (only supported with TRE regex, not supported in GNU regex.)

`<fromstart>` - start looking for a matching string at the start of the available text (this is the default; use it if you want to emphasize to those who might read your program that you are starting from the front of the examined string in a program that often uses other starting search flags)

`<fromcurrent>` - start looking for a matching string at start of previous successful match on the available text. Thus, any text that was not matched by the most recent previous match on this variable will not be eligible for matching.

`<fromnext>` - start looking for a matching string at the next character past the start of the previous successful match on the available text. This means that the match result of this match will always be at least one character further down the string, and perhaps much further, if it succeeds at all.

`<fromend>` - start looking for a matching text at one character past the end of the most recent previous successful match on this text. This guarantees that no character in the previous successful match will be in the new match.

`<newend>` - require the new match to end after the end of the most recent previous successful match on this text. The found match will start at a character after the most recent successful match's first character.

`<backwards>` - search backward in the available text from the most recent successful match, rather than forward. This is somewhat slower than forward matching.

`<nomultiline>` - searches the available text in chunks of one line each, thus ^ will match at the start of each line an **`$`** at the end of each line; this also implies that the pattern cannot span more than one line. This doesn't mean the match fails if the pattern isn't found on any particular line, just that matching will be keep trying on successive lines till it either finds the pattern, or fails (in which case, the MATCH statement skips to the end of the block like a FAIL statement does). Without this flag, the available text is searched as a single block, so ^ only matches at the start of the entire text and **`$`** only matches at the end of the entire text.

Note that **`<nomultiline>`** does not interact with **`<absent>`** to give a "search for the first line that does not contain this pattern", it means "if this pattern cannot be found anywhere in the search area, succeed, otherwise fail". If you need "first line not containing, skip down to “Idioms and Tricks: Finding Lines that Don't Contain”.

Variable Restrictions

Sometimes you want to execute the match against a string other than **`:_dw:`** , or possibly you want to match against a substring that starts at a given point, with a given length, or even possibly against a string matched within a string. This is done with a *variable restriction*. Variable restrictions are the “indirect object” nouns in the CRM114 language- typically the variable restriction provides the context for the statement to operate in.

A variable restriction looks like this;

```
      [ :some_place:  optional_starting_char  optional_length ]
```

and it can be used at the statement level in many CRM114 statements. In particular, a variable restriction can tell a match where to search, an input or output statement what filename to read or write (and at what byte offset, and how many bytes), or a CLASSIFY statement what text to classify. Many statements in CRM114 can use a variable restriction- see the actual statement descriptions for each statement for details.

How to Change What Variable To Use

The simplest use of variable restrictions is to change the variable that a particular statement will be getting input from. This is done with just the variable name inside the square boxes – just add the restriction to the statement. Here's an example- we'll go from matching for **`/foo bar baz/`** in the data window, to matching on it in a variable named **`:my_var:`**;. Here's the match in **`:_dw:`**

```
       match /foo bar baz/
```

and here it is, in **`:my_var:`** instead:

```
       match /foo bar baz/ [ :my_var: ]
```

This is the simplest form of variable restriction.

How to Use a Substring, by Start / Length

We can make variable restriction change not just what variable to use as input, but also what part of the variable. For example, we can subscript out a section of a variable, giving a starting character index, and then how many characters of text we want to use.

To do this subscripting of a part of a variable, use the following three pieces in order inside the `[]` variable restriction boxes:

1. "what variable to use". This must be the name of a variable.
2. "starting at what character" . This is the first character to use. All counts are zero-based (that is, the 0th character is the first character in a string or file.
3. "how many characters to use". This is the 'inclusive' count- that is, any string 0 characters long contains nothing at all. The string "a" contains just one character (you don't need to ever worry about a trailing NULL or other petty annoyances in CRM114).

If you specify a starting character, but no length, the length extends to the end of the string.

You can't specify a length without specifying a starting character, so to get the first N characters of a string you have to use 0 as the starting point.

Here's a silly example of a variable restriction on a match – we want to see if the word "foo" exists somewhere in the block starting at the 10th character (zero-based addressing, remember) and five characters long of the input. Here's the code:

```
{
      match [ :_dw:  10 5 ]  /foo/
      output /Yes, there is one \n/
}
```

When we run this code, we get:

```
# crm match_restr.crm
foo bar baz wugga
return, then ctrl-D
# crm match_restr.crm
alpha foo bravo
return, then ctrl-D
# crm match_restr.crm
alpha bravo foo charlie
return, then ctrl-D
Yes, there is one
#
```

Further experiments will demonstrate to you that the "Yes, there is one" message is only printed out when there is a "foo" somewhere in characters 10 through 14 of the input text.

Remember – CRM114 uses zero-based indexing, so the initial character of a string is the zeroth character and the character at index 1 is the second character!

How to Use a Substring, by Regex

A variable restriction can also be based on a regex itself. This is effectively applying a regex to the variable, and using the part of the string that matched as the input to the "main" regex text

Restriction regexes are a way to let a statement choose its own input from a string. The variable restriction regex looks a lot like the subscripting regex, that is if you are using a MATCH on your variable named **`:my_var:`** like this:

```
match [ :my_var: ] /foo bar baz/
```

and realized you wanted only to match inside **`:my_var:`** between the keywords **`alpha`** and **`omega`** then you could add the restriction regex like this:

```
match [ :my_var: /alpha.*omega/ ] /foo bar baz/
```

that would start with :my_var: and then look for the string that matched **`/alpha.*omega/`** inside that variable, and then used the result of that match as input to the main match for the **`/foo bar baz/`** regex.

Differences between a Restriction Regex and the Main Regex

The differences between a restriction regex and the main regex in a match are

- The main regex in a match can FAIL, but a restriction regex can't cause a FAIL. If the restriction regex doesn't match, then the main regex is given a zero-character string. If the main regex can match a zero-character string (such as with a zero-length alternation, as **`/foobar|()/`**, then the MATCH statement won't FAIL)

- The restriction regex doesn't affect the previous-match information (such as **`<fromnext>, <fromend>`**, etc) , unlike the main regex.

- The restriction regex is not affected by previous-match values, unlike the main regex. If a previous-match modifier (like **`<fromend>`**) is present, then the combined effect is that only the substring eligible for match that was accepted by both the restriction regex and the previous match modifier is input to the main regex.

- The restriction regex result is the last parenthesized subexpression that returned a partial match. (hint: parenthesized subexpressions return a partial match unless they start out with **`'(?:'`** as a preamble instead of **`'('`**). If there are no parenthesized subexpressions then the whole restriction match result is fed to the main regex.

This makes it easy to find particular substrings by preamble/postamble; put the regex expression that matches the text you actually want into the deepest-parenthesized subexpression, and specify the preamble and postamble as desired.

In the theoretical sense, variable-restriction regexes can't do anything that can't also be done with a either a suitable outer ornamentation of the main match regex or the use of an intermediate MATCHed variable to carry information from one MATCH to the next. However, there may be engineering reasons for using a variable-restriction regex:

- A variable-restriction regex is somewhat faster to execute than by creation of an intermediate variable, especially if the variable will be used only once. This is because intermediate variable creation causes CRM114 to do bookkeeping on the variable's name; there is no bookkeeping required with a var-restriction regex. Data in the intermediate variable must be checked for reclamation hold back when the outer variable is reassigned (which is low cost but not nonzero cost); the var-restriction regex never creates any intermediate variable and so it never impacts space reclamation.

- Redecorating an already-complex main regex may make it too complicated to understand.

- The set of characters available for matching after variable restrictions are ANDed with the set of characters available for matching with match control flags like **`<fromstart>, <fromnext>`** , etc. If there are no characters that simultaneously satisfy both requirements, the main regex gets the empty string.

- Variable restriction regexes and subscript-based variable restrictions can be pipelined.

The concept of *pipelining* a variable restriction bears further explanation.

Pipelined Variable Restrictions

The CRM114 system allows a series of variable restrictions to be connected end-to-end; the output of each restriction is used as the input to the next restriction. There's no special syntax or **|** pipe character required; just keep whitespace between the variable restrictions.

Both regex-based and subscripting-based variable restrictions can be freely intermingled in the restriction pipeline.

- Single numbers are considered to be starting offsets and the rest of the string is used.
- Pairs of numbers are used as start/length restrictions.
- Strings in slashes like **`/foo/`** are used as regex restrictions.

Here's an example; we'll pipeline two regexes around a start/length subscript operation, and use .* for the main regex, so everything accepted by the variable-restriction pipeline will be in the final match.

```
{
      match [:_dw:  /abc.*xyz/  4 20 /[[:alpha:]]+/] /.*/ (:my_stuff:)
      output /Found my_stuff as ":*:my_stuff:"\n/
}
```

The regex pipeline here finds the first-then-longest occurrence of **`abc`** followed by any arbitrary characters followed by **`xyz`**. A chunk of text is then subscripted out, starting at the fifth character (zero-base, remember), and extending 20 characters. Whatever survives that subscripting is then searched for the first run of one or more alpha characters (that's a-z and A-Z only, nothing else allowed). Whatever survives *THAT* is then matched with the main regex **`.*`** (the regex **`.*`** matches anything – and incidentally, "anything" includes "the zero length string containing nothing") and is printed out. When run, we get:

```
      # crm match_pipeline.crm
      zebra abc silly thing to look for xyz giraffe
      Found my_stuff as "silly"
      #
```

showing that the convolutions actually do function.

Whether to use intermediate variables or a regex variable restriction in any particular situation is not a simple problem. Restriction regexes are slightly faster, especially in a loop. Restriction regexes work in conjunction with the match control flags, and intermediate variables don't. Sometimes, the restriction regex can make the situation clearer; sometimes it hurts understandability.

Use restriction regexes with care.

LIAF and FAIL

So, we now have MATCH, which can succeed or fail depending on whether the regex matched or not, detouring execution to the end of a **{}**-block. Symmetry would require some easy way to go to the *start* of a **{}**-block; this is exactly what LIAF[11] does. For symmetry, should we need to go to the end of a **{}**-block; the FAIL statement does this.

Iterative Looping with LIAF

The commonest use of LIAF is to express iterative looping, that is, repeating the same code over and over again. Typically, the LIAF is used just before the end of the **{}**-block, to cause the block to re-execute. To prevent an infinite loop, you should use a MATCH that will fail when the iteration is complete and block should be exited. Fortunately, this is almost always “free” - the same match that starts the work of the block can also tell when to terminate the block.
The syntax of LIAF is simple:

```
        liaf
```

and it takes no parameters or flags.
Here's an example program that prints out every line of the input that contains the letter 'a', and only those lines.

```
        {
                match <nomultiline fromend> (:my_line:) /.*a.*/
                output /Found an 'a' here --> :*:my_line:\n/
                liaf
        }
```

Notice that the MATCH statement uses two flags – **nomultiline** and **fromend** . This forces the repeated iterations to always start on a new input line, and to consider each line of input separately, not with all of the input as a single big block (single big block is the default). The match pattern also uses two wildcards: **.*** both before and after the **a** – this means that the full match will be the entire line where an **a** was found.

Every time the match succeeds, the OUTPUT statement will execute, printing out the line that contained a letter 'a'. When we run out of lines containing 'a', the MATCH will fail, and the **{ }** block will exit (ending the program).

11 Because the commonest use of LIAF is in iteration, LIAF means Loop Iterate Awaiting Failure. If that's too hard to remember, just pretend that LIAF is FAIL spelled backwards.

When we run this program, we get:

```
# crm find_the_a.crm
one two three
this is a test
does this line have the magic letter?
I suppose it did, but this one doesn't.
one more line for luck.
Found an 'a' here --> this is a test
Found an 'a' here --> does this line have the magic letter?
#
```

So, now we have a program that is beginning to mimic `grep` (the text-finding program available in most *NIX systems)- assuming you only want to find the letter 'a'. But finding just the letter 'a' is not particularly interesting or useful. We can modify our code to be more grep-like, by having it obtain the regex pattern from the command line as a *command line argument*.

Using A Command Line Argument

When run from the command line as a **`#!`** -program, CRM114 execution engine's name is put into **`:_arg0:`** your program's name is **`:_arg1:`**, and the variables **`:_arg2:`**, and so forth are your command-line arguments. This is easy to use- we'll just use **`:_arg2`** as the regex pattern to look for.
Here's the program:

```
#!/usr/bin/crm
output /Looking for lines with a ':*:_arg2:' \n/
{
        match <nomultiline fromend> (:my_line:) /.*:*:_arg2:.*/
        output /Found an :*:_arg2: here --> :*:my_line:\n/
        liaf
}
```

Notice that the MATCH pattern is `.*:*:_arg2:.*` This regex looks daunting- let's take it apart.
Remember when a pattern is var-expanded, first **`\`**-constants are expanded (here, there are none). Then, **`:*:`*`var`*`:`** vars are expanded; in this case, **`:*:_arg2:`** is expanded to the text of the first user argument on the command line. Then repeated MATCH / OUTPUT / LIAF looping happens till we run out of input.

Here's the actual program output, using **`lollipop`** as the command-line argument:

```
# crm my_grep.crm lollipop
```

```
hello there.
Do you have a lollypop?
How about a lollipop, spelled with an 'i'?
And are there any pickles?
I like lollipops.  Please give me one.
return, then ctrl-D
Looking for lines with a 'lollipop'
Found an lollipop here --> How about a lollipop, spelled with an
'i'?
Found an lollipop here --> I like lollipops.  Please give me one.
#
```

Our LIAF-loop worked perfectly. Only the lines containing our command-line argument of “lollipop” were printed.

Notice that the starting OUTPUT line – telling us we were looking for 'lollipop' – did not occur till after the entire input had been read in. This is normal CRM114 behavior – read all of `stdin` until EOF, then start execution. Usually this is the desired behavior, but if you need to change this behavior, read on under the WINDOW statement.

Going to the end of block with FAIL

Occasionally, it's useful to simply exit the current **{}**-block; sometimes it's because your heuristic has dead-ended and it's time to try a different approach. You *could* just execute a MATCH that you know will fail, but that lacks grace (and means that the next person trying to debug your code will go crazy trying to understand why you have a MATCH statement with a regex that makes no sense).

In cases like this, it's better to use the FAIL statement. FAIL exits the block, and sets the block exit status so that an ALIUS statement will know the block FAILed (in contrast to an ALIUS statement itself, which SKIPs to end of block but does not set failure.)

The syntax of FAIL is simply;

```
fail
```

and it takes no parameters or flags.

Here's a simple example using a FAIL statement:

```
{
        output /Starting here.../
        {
              output /in the inner block, about to FAIL.../
              fail
              output / still in inner block— You shouldn't see this /
        }
        output /back to the outer block \n/
  }
```

When we run this example, we get:

```
# crm fail_1.crm
return, then ctrl-D
Starting here...in the inner block, about to FAIL...back to the
outer block
#
```

which shows that FAIL did in fact end execution of the inner block.

The ALIUS statement

When a match or a block FAILS, the FAILed block itself returns a hidden value in CRM114, and we can use this value for flow of control, with the ALIUS[12] statement. ALIUS acts like an ELSE clause on the most recently closed-out **{}**-block. If the most-recently closed-out **{}**-block exited with a FAILing statement, then the ALIUS acts as a no-op, allowing execution to continue into the next statement at the current level. If the most recently closed-out **{}**-block exited "normally" (that is, without FAILing) then the ALIUS statement does a SKIP[13] to the end of the current **{}**-block. This is a SKIP, not a FAIL, and subsequent superior **{}**-blocks will not see a failure exit status from this block.

The syntax of ALIUS is just:

```
alius
```

ALIUS as an ELSE statement

Here's a program demonstrating ALIUS by looking for a 'zebra' in the incoming text:

```
{
      {
            match /zebra/
            output /Found a zebra! \n/
      }
      alius
      {
            output /There are no zebras in here. \n/
      }
}
```

12 *alius* (pronounced ahh-lee-oos) is from Latin, meaning "the other", "the other man", or "alternatively"

13 There is currently no unconditional SKIP statement in CRM114. That's because nobody has shown any need for it. Symmetry reasons say we should have a SKIP, parsimony says we shouldn't have it. Correctness proofs or good examples of either will be gladly accepted, and in the interim, should you ever need to SKIP, remember that a TRAP statement with a regex that never matches anything will have the same effect as the nonexistent SKIP statement.

Running our zebra-seeking program, with text containing a zebra:

```
        # crm alius_zebras.crm
        breakfast cereal
        zebra
        ostrich
        return, then ctrl-D
        Found a zebra!
        #
```

which behaves exactly as we would have expected- there was a zebra in the input, so the program printed "Found a zebra!" Running it again, without a zebra in the input:

```
        # crm alius_zebras.crm
        lions
        tigers
        bears
        giraffes
        snakes
        return, then ctrl-D
        There are no zebras in here.
        #
```

ALIUS as a CASE or SWITCH statement

ALIUS statements always work on the most recently closed-out **{}**-block, so you can use ALIUS as a CASE or SWITCH statement. Here's an example of ALIUS casing, with four alternatives (lions, tigers, bears, and none-of-the-above). Each of the cases starts out with a MATCH to determine if this is the proper case to execute:

```
        {
              {
                    match /lion/
                    output / Lion roars! \n/
              }
              alius
              {
                    match /tiger/
                    output / Tiger leaps! \n/
              }
              alius
              {
                    match /bear/
                    output / Bear growls! \n/
```

```
            }
            alius
            {
                  output / This place is boring.  Ho-hum... \n/
            }
      }
```

Let's run it a few times.

```
      # crm multiple_alius.crm
      I've heard that lions are kingly beasts
      return, then ctrl-D
       Lion roars!
      # crm multiple_alius.crm
      I've heard that bears ... something about the woods.
      return, then ctrl-D
       Bear growls!
      # crm multiple_alius.crm
      And I'd wonder about tigers.  Don't they bounce?
       return, then ctrl-D
       Tiger leaps!
      # crm multiple_alius.crm
      Are there any other animals here?
      return, then ctrl-D
       This place is boring.  Ho-hum...
      #
```

As we can see, the first **{}**-block that matches successfully ends the ALIUS chain. We don't need another SWITCH, CASE, ELSE, GOTO END or a BREAK like C; the ALIUS statement alone suffices.

Formally, ALIUS skips to the end of the current block (that's a SKIP, not a FAIL) if the most recently exited block exited successfully; if the most recently exited block itself exited because of a FAIL, then ALIUS is a no-op (and execution continues with the next statement after the ALIUS). This is exactly the behavior that allows easy construction of IF-THEN-ELSE and serial CASE program structures.

One should note that a successful ALIUS termination of a block does <u>not</u> cause the next outermost block to see a failure – the successful ALIUS termination is a SKIP to end of block, not a FAIL.

If you want the final ALIUS of a block to cause a FAIL back up to the next enclosing block, you must do an explicit FAIL in your code, not inside the nested clause.

Here's an example- note that the fail is <u>not</u> wrapped inside a `{}`-block:

```
      {
            {
                  {
                        match /rugby/
                        output / found the word “rugby” \n/
                  }
                  alius
                  fail           ####  THIS FAIL IS NOT INSIDE A BLOCK!!
            }
            alius
            output / We failed successfully as final ALIUS clause \n/
      }
```

When run, we get:

```
      # crm alius_fail.crm
      Do you ever watch rugby tournaments?
      return, then ctrl-D
       found the word “rugby”
      # crm alius_fail.crm
      No, actually I prefer gaelic football.
      return, then ctrl-D
       We failed successfully as final ALIUS clause
      #
```

showing that we can indeed propagate the FAIL status of an ALIUS chain up to the next enclosing block, but we have to do it explicitly. Without the FAIL, we have:

```
      {
            {
                  {
                        match /rugby/
                        output / found the word “rugby” \n/
                  }
                  alius
                  ####fail     ####  THIS FAIL IS NOT INSIDE A BLOCK!!
            }
            alius
            output / We failed successfully as final ALIUS clause \n/
      }
```

which simply does a SKIP to the close of the `{}`-block; the block then exits as “successful”, and the outer ALIUS then SKIPS again:

```
# crm alius_not_fail.crm
Although rugby is fun, the final ALIUS clause does not fire off.
return, then ctrl-D
 found the word “rugby”
# crm alius_not_fail.crm
And we haven't even mentioned the game of 'hurling'.
return, then ctrl-D
#
```

showing that the we indeed needed the inner FAIL in order to tell the outer ALIUS to keep trying clauses.

GOTO and Statement Labels

CRM114 provides the much-maligned GOTO statement. The GOTO syntax is simply:

```
goto /:statement_label:/
```

where `:statement_label:` is the statement label, on a line by itself (yes, a statement label is a full statement in CRM114). The statement label value is var-expanded with math enabled, so you can use **`\`**, **`:*`** , **`:+`** , and **`:@`** expansions with the GOTO location.

It's common to forget that the CRM114 GOTO statement is **`/slash-demarcated/`**. Just coding **`goto foo`** will not work! You'll get an error like:

"This program has a GOTO without a place to 'go' to. By any chance, did you leave off the '/' delimiters?"

To avoid embarrassment in front of your co-workers, remember that everything in CRM114 except command words and comments must be marked with delimiters.

Here's an example program that uses GOTO and statement labels:

```
{
      output /start here.../
      goto /:jump_over:/
      output /You should not see this\n/
      :jump_over:
      output /and finished. \n/
}
```

When run, we get:

```
# crm goto_1.crm
return, then ctrl-D
start here...and finished.
#
```

demonstrating that we did GOTO the label `:jump_over:`, as "You should not see this" never ran.

You can also GOTO a particular line number in a program, but it is EXTREMELY UNWISE to do this to hard-coded line numbers. The ability to GOTO a line number is only reasonable when used with the **`:_cs:`** current statement variable, in association with a FAULT or TRAP fix up. If you hard-code a line number, you will very quickly find your code becomes unmaintainable as new lines are added and deleted..

GOTOs have occasionally been termed the last refuge of an incompetent programmer.   Be concerned that you might be writing bad code if you find yourself using a tangled web of GOTOs. On the other hand, a correct use of GOTOs might be if the problem supports a lattice-type solution; in this case a stream of forward GOTOs is the most elegant solution.

In general, let beauty be your guide; symmetrical code is almost always easier to write and to debug.

I/O with the INPUT and OUTPUT statements

So far, we've always used CRM114's automatic read-to-EOF to input any necessary data, and we've always outputted to standard output. This is adequate for many programs, but if you need to read or write into the file system by filename (rather than as controlled by redirection) you will need to use INPUT and OUTPUT statements.

The CRM114 input and output statements are capable of doing random-access I/O to anything in the filesystem namespace that the user has access rights for. Although `ioctl` operations are not supported directly in the language, fseek and block I/O are supported; in fact, they're integrated into the I/O statements.

Here are some basic syntax variations on INPUT (default input var is into **`:_dw:`**, and default file to read is `stdin`). You may notice that the file specification looks a lot like a subscript-style variable restriction, with the start and length specifications; this is intentional.

```
      input
      input (:var_to_get_input:)
      input (:var_to_get_input:) [ file_to_read ]
      input (:var_to_get_input:) [ file_to_read lseek_here]
      input (:var_to_get_input:) [ file_to_read lseek_here byte_count ]
```

and similarly for output (but note- there is no default output string, you must always specify it):

```
      output /string_to_output/
      output /string_to_output/ [ file_name ]
      output /string_to_output/ [ file_name lseek_here ]
      output /string_to_output/ [ file_name lseek_here byte_count ]
```

In CRM114, there is no concept of an open file, a file handle, or a file descriptor (remember, with CRM114 it's "strings all the way down"). All file I/O is done in single-statement operations, and the I/O operation is done as a single block transfer of the full contents (giving very high performance compared to repeated calls to `fscanf` or `fprintf`).

A CRM114 statement to do INPUT or OUTPUT actually does the following:

- if the file is not `stdin` or stdout, then `fopen` the file for the appropriate access mode
- if necessary, execute an `fseek` to the appropriate starting location in the file
- execute an `fread` or `fwrite` (then `fflush)` of the appropriate length (default is to read or write the entire file at once)
- `fclose` the file.

This minimizes contention on file locking (but if you're doing random I/O to shared files and the possibility of multiple simultaneous access exists, don't assume that everything will "just work" - you will need to use another, more definitive file locking mechanism unless your filesystem itself can support an N-readers-or-1-writer mutual exclusion.)

To see how fast this can go, let's benchmark an example of how fast CRM114 can do large-block I/O. This program will read in the entire `/usr/share/dict/linux.words` dictionary file (about 4.9 megabytes), and output it into a new file, `cloned.words` . We'll use the `time` command to measure the elapsed time.

Two things to notice – first, the INPUT statement will default to putting the input in **`:_dw:`** , which is the 'default place'. Unlike INPUT, the OUTPUT statement has no such default, so we do have to specify **`:_dw:`** in the OUTPUT statement. In this case, the CRM114 paradigm of "when in doubt, nothing out" overrides the paradigm of **`:_dw:`** as the default place.

Here's the dictionary-copying source code; it copies from `/usr/share/dict/linux.words` to `cloned.words` in the current directory (please forgive the WINDOW statement; for now, assume that it simply means "don't wait for EOF on `stdin`" which is enough to know for now):

```
{
      window
      input [ /usr/share/dict/linux.words ]
      output [ cloned.words ] /:*:_dw:/
}
```

Here's the results (we're using the bash command `time` to get actual execution time)

```
# time crm clonedict.crm

real    0m0.324s
user    0m0.153s
sys     0m0.150s
#
# ls cloned.words  -la
-rw-r--r--  1 root root 4992010 Dec 24 12:44 cloned.words
#
```

This performance is not too shabby, especially for a laptop computer. CRM114 moved 5 megabytes in about 0.4 seconds, and that's on a Transmeta 933 subnotebook, and includes compilation time. For comparison, here's `cat` in the same task :

```
# time cat < /usr/share/dict/linux.words > cloned.words2

real    0m0.215s
user    0m0.013s
sys     0m0.124s
#
```

showing that CRM114 is not unreasonable for doing large amounts of I/O. (for comparison, the **`cp`** copying program can do this in 0.127 seconds total elapsed time.)

Random-Access File I/O

CRM114 can do random-access I/O using the same offset / length syntax as is used for a variable restriction; the only difference is that it's applied to the filename. The important parameters are the filename, the start position (which is equivalent to an `lseek` from the start of the file) and the length (equivalent to the number of bytes to be transferred)

The basic syntax is :

```
input [ filename start_pos number_of_bytes ] (:destination_var:)
```

and for output

```
output [ filename start_pos number_of_bytes ] / var_expanded_text /
```

In both of these statements, the entire **`[]`** var-restriction is optional; if no var-restriction is given, then the default is to do I/O to standard input and output (`stdin` and `stdout`, respectively).

Additionally, for output, you can specify `stdout` or `stderr` if you want to output to those streams. Since `stdout` is the default, there's usually no reason to specify it, but **`output [stderr] /some_info/`** can be handy to output debugging or error information to the user terminal.

If a filename is supplied, it is optional to also supply a start position. If the start position is supplied, it is optional to supply the number of bytes to transfer.

All parts of the `[]`-restriction are given full var-expansions - `\`-constants, `:*` variable substitutions, `:#` string-length substitutions, and `:@` math evaluations are all evaluated. Unlike a full MATCH-style variable restriction, you can't use a regex in an INPUT or OUTPUT var-restriction.

Here's an example, where we read in 5 characters, starting at character number 10 from file `foo.txt`, and print out those 5 characters. Remember that the starting offset is zero-based – the first character is character zero, and character number 10 is actually the *eleventh* character.

```
{
      input [foo.txt  10  5]
      output /The five characters are ':*:_dw:'\n/
}
```

Putting a standard message into the `foo.txt` file, this example program results in:

```
# cat > foo.txt
The quick brown fox jumped over the lazy dog's back 1234567890
# crm random_input.crm
return, then ctrl-D
The five characters are 'brown'
#
```

Reading One Line at a Time

It's possible to read a single line at a time from standard input (`stdin`).   This is set by using the **`<byline>`** flag on the INPUT statement. The good news is that this makes perfect sense on `stdin` but the bad news is that since ordinary files are always freshly opened when read, repeatedly using **`<byline>`** on an ordinary file will get you the first line of the file, over and over again. Fear not- you can get line-at-a-time behavior *plus* the speed advantages of block IO by using the WINDOW statement, so read on to the chapter on WINDOW.

Here's a program that will read three lines after the ctrl-D EOF and print them out.

```
{
      input <byline>
      output /Line #1 = :*:_dw:\n/
      input <byline>
      output /Line #2 = :*:_dw:\n/
      input <byline>
      output /Line #3 = :*:_dw:\n/
```

```
        }
```

Note that we still have to give it an EOF to start it running. Here's the result:

```
        # crm byline_reads.crm
        return, then ctrl-D
        Alice asked the cheshire cat
        Line #1 = Alice asked the cheshire cat
        just why it seemed to be fading
        Line #2 = just why it seemed to be fading
        but the cat replied in no clear terms
        Line #3 = but the cat replied in no clear terms
        #
```

Of course, we could have used a LIAF loop and captured lines forever, but that would be boring unless we had some processing to do on the lines that we wanted (there's a programming hint there!)

Something to be careful of with INPUT. All input, even **`<byline>`** input, is terminated on EOF (that is, the I/O completes and the statement finishes, rather than waiting for more input). This means that if you write a program that repeatedly does **`<byline>`** input on an empty file, it will loop at a very high speed waiting for a full line to be entered. Programs that do this kind of input will need some kind of auxiliary throttling to prevent a CPU-sucking loop. We recommend a

```
        syscall / sleep 1 /
```

which will put your program to sleep for one second; you can accept this on faith for now. You'll see more of this in the chapter on SYSCALL below.

Specialized I/O with ACCEPT and WINDOW

INPUT and OUTPUT are the main I/O operations in CRM114. There are two other statements that also do direct I/O; these are ACCEPT and WINDOW.

ACCEPT: The Short And Shorter Of It.

ACCEPT is a shortcut; it takes no flags or parameters, and the syntax is just:

```
accept
```

ACCEPT is entirely equivalent to

```
output /:*:_dw:/
```

but perhaps a little more obvious. Its use is mainly in go/no-go filters, to emphasize that the output is exactly the input.

Of all of the statements in CRM114, ACCEPT will be the first up against the wall when the revolution comes.

WINDOW – Chunking Through Your Data

WINDOW is a somewhat complex statement, because it WINDOWs a variable through an input stream (including a potentially infinite input stream), on “chunks” delimited by whatever a regex matches. WINDOW is also the way a program can override the default CRM114 read-to-EOF behavior.

WINDOWing to Start Execution Without Read-To-EOF

WINDOW, by itself (no options) as the first executable statement of your program turns off the read-to-EOF default; this lets your programs “come out running”. The good news is that this lets your programs print out banners, etc. The bad news is you have to do your own input.

Here's a simple example:

```
{
      window
      output / Notice we didn't have to hit ctrl-D ! \n/
}
```

And it works:

```
# crm window_1.crm
 Notice we didn't have to hit ctrl-D !
#
```

So, at last, you now know how to un-require ctrl-D. Note that this works only if WINDOW is the first executable statement of a CRM114 program.

WINDOWing through a long input

If you WINDOW with no parameters to prevent CRM114 from reading standard input to EOF before beginning, then obviously your program will need to obtain its data in some other way. A regular INPUT statement will work, but by default INPUT reads an entire file at a time. If that doesn't suit your needs, you need WINDOW. INPUT also requires some machinations to read past EOF on a growing file sequentially.

WINDOW, on the other hand, is meant to gracefully work its way through a (possibly infinite, possibly growing file), with a minimum of fuss.

WINDOW uses two regex strings, up to two variables, and has several optional flags.

WINDOW execution works in two phases: cut and add. The first "cut" phase removes a controlled amount of text from the first variable with the first regex:

- The first parenthesized variable in the statement is the "windowed" variable. By default, this is **`:_dw:`**, the default data window (now you see why it's called the 'window', eh?)
- The first regex in the statement is the "cut" regex. Everything in the windowed variable up to the first match of the "cut" regex is cut from the windowed variable and thrown away.

The second phase is the "add" phase, and uses the windowed variable, the second "source" variable, and the second regex:

- The second parenthesized variable is the "source" variable. By default, this is `stdin`, although you can use any variable you want (though we recommend that you don't use **`:_dw:`** as the source variable if you also are using **`:_dw:`** as the windowed variable. The same applies to any other use of the same variable as the source and windowed variables.)

- The second regex is the "add" regex. Everything from the start of the source variable up to and including this "add" regex is cut from the source variable and appended to the end of the windowed variable. Note that the "add" regex does not need to be the same as the "cut" regex.

So, WINDOW partitions a big block of text (either already INPUTted into a variable, or right from `stdin`) and puts it in convenient bite-sized chunks in another variable. Every invocation of WINDOW does another such chunk.

There's always the possibility that the input never matches the "add" regex; in this case WINDOW acts just like MATCH and does a FAIL to the end of the current **{}** block.

Here's an example – let's say that our chunk of interest is a 'word'- that is, a string of non-space characters broken up by spaces. We want to break up a text and see the individual words, one at a time. Here's the code:

```
{
      window    #  this WINDOW turns off the default read-to-EOF
      {
           window  /.*/      / /     #  Yes, one single space.
           output / Here's a word: ':*:_dw:' \n/
           liaf
      }
      output / All done! \n/
}
```

The result is:

```
# crm window_2.crm
Beware the Jabberwock, my son.  <-- hidden trailing space before return.
 Here's a word: 'Beware '
 Here's a word: 'the '
 Here's a word: 'Jabberwock, '
 Here's a word: 'my '
 Here's a word: 'son. '
 Here's a word: ' '
now hit ctrl-D
 All done!
#
```

If we didn't have that trailing space, then the "add" regex would never have been satisfied for the string "son.", so the WINDOW statement did a FAIL and exited the block, so we'd never see the “son.” in the output stream.

Here's that no-space case:

```
# crm window_2.crm
Beware the Jabberwock, my son.  <-- note no trailing space here, just return
 Here's a word: 'Beware '
 Here's a word: 'the '
 Here's a word: 'Jabberwock, '
 Here's a word: 'my '
hit ctrl-D
 All done!
#
```

We can fix that with the **`<eofaccepts>`** flag on the WINDOW statement. The **`<eofaccepts>`** flag allows any EOF to be counted as equivalent to a successful add-regex match. Here's the slightly-modified code:

```
{
      window    #  this WINDOW turns off the default read-to-EOF
      {
            window  <eofaccepts> /.*/   / / # Yes, one single space.
            output / Here's a word: ':*:_dw:' \n/
            liaf
      }
      output / All done! \n/
}
```

which works “better” - but not perfectly. Here's the result:

```
# crm window_3.crm
Beware the Jabberwock, my son.   <-- no trailing space, just return
 Here's a word: 'Beware '
 Here's a word: 'the '
 Here's a word: 'Jabberwock, '
 Here's a word: 'my '
hit ctrl-D here.
 Here's a word: 'son.
'
 Here's a word: ''
 Here's a word: ''
 Here's a word: ''
 Here's a word: ''
 Here's a word: ''
```

```
.....
hit ^C here.  It's an infinite loop.
#
```

This is "better", but now since the WINDOW will never do a FAIL, we need to provide some other way for the LIAF-loop to end. In our example here, a simple match to let us know we had at least one character in the WINDOW is sufficient. Here's the code

```
{
      window    #  this WINDOW turns off the default read-to-EOF
      {
            window  <eofaccepts> /.*/   / / # Yes, one single space.
            match /./
            output / Here's a word: ':*:_dw:' \n/
            liaf
      }
      output / All done! \n/
}
```

and it works as desired, yielding single words:

```
# crm window_4.crm
Beware the Jabberwock, my son. <-- hit return here
 Here's a word: 'Beware '
 Here's a word: 'the '
 Here's a word: 'Jabberwock, '
 Here's a word: 'my '
hit ctrl-D here
 Here's a word: 'son.
'
 All done!
#
```

which is correct (the "misplaced" single quote after "son." is correct; after all, there is a newline in the input file at that position.).

You might think that perhaps it would be better to use a regex that includes our definition of a word, rather than a word boundary. That works- but not the way you probably expect. Consider that a typical definition of a "word" is a consecutive set like **`[a-zA-Z]+`** . If you try this as the regex, in code like this:

```
{
      window    #  this WINDOW just turns off the default read-to-EOF
      {
            window  /.*/   /[a-zA-Z]+/
            output / Here's a word: ':*:_dw:' \n/
            liaf
      }
      output / All done! \n/
}
```

then you get the mildly unsettling:

```
      # crm window_5.crm
      Beware the Jabberwock, my son.  <-- hit return, no trailing space
       Here's a word: 'B'
       Here's a word: 'e'
       Here's a word: 'w'
       Here's a word: 'a'
       Here's a word: 'r'
       Here's a word: 'e'
       Here's a word: ' t'
       Here's a word: 'h'
       Here's a word: 'e'
       Here's a word: ' J'
       Here's a word: 'a'
       Here's a word: 'b'
       Here's a word: 'b'
       Here's a word: 'e'
       Here's a word: 'r'
       Here's a word: 'w'
       Here's a word: 'o'
       Here's a word: 'c'
       Here's a word: 'k'
       Here's a word: ', m'
       Here's a word: 'y'
       Here's a word: ' s'
       Here's a word: 'o'
       Here's a word: 'n'
       All done!
      #
```

What's gone wrong here is that WINDOW, in order to make reasonable chunks out of the input, must take the first (meaning minimal) match from the source. So, although **`Beware`** does match the regex **`[a-zA-Z]+`**, it's also true that just **`B`** matches as well.

So, important safety tip:

The regexes used in a WINDOW should match the boundary of the desired chunks, not the entire chunks. Otherwise, the “minimum match” effect may cause your WINDOW chunksize to be far smaller than you expected.

If you would prefer to match entire chunks, use a LIAF-loop instead of a WINDOW loop.

That said, there are a few options for WINDOW that may come in handy:

`<nocase>` - works just as it does in a MATCH statement – it ignores case in the regex matching.

`<bychar>` - read one character at a time when reading from `stdin`. This option is a little slow, but gives the right behavior when you don't want to read too much data (such as when another program will also be reading `stdin` when the CRM114 program finishes, and so it becomes important that the CRM114 program not use up any `stdin` characters it shouldn't.)

`<bychunk>` - read in a conveniently large chunk from `stdin`. This is often all the way to EOF, but not necessarily so. If more data is needed, another chunk is read.

`<byeof>` - read `stdin` all the way to EOF (or at least as much as possible within the anti-DoS limits), and buffer the unused data until it's needed by another WINDOW statement.

`<eofaccepts>` - as seen in the example above, this allows an EOF to be accepted as a “match” for the add regex. Be warned- if you use this you will need to provide another way throttle looping and exit your processing or you will loop forever.

`<eofretry>` - instead of failing on EOF, this causes EOF to be reset and the failing read operation to be retried. This also can cause rapid looping; use with care.

WINDOWing from or to a variable or a file

By default, WINDOW reads from `stdin`, and windows the data window **`:_dw:`** . For many programs, this is adequate, but sometimes you'll want to read from another variable or even from a file.

CRM114 does not provide any direct way to WINDOW from a file (as that would imply a file being kept open for long periods of time). Instead, the recommended method is INPUT the file into an ISOLATEd variable and then use that variable as the second (source) variable in the WINDOW statement. Let's make this clearer with some code; we'll pull in the `bash` FAQ text file, and WINDOW our way through it. Here's the code:

```
      {
              window   #  this WINDOW turns off the default read-to-EOF
              input (:bash_words:) [/usr/share/doc/bash-3.0/FAQ]
              {
                      window  /.*/   / /  (:_dw:) (:bash_words:)
                      match /./
                      output / Here's a word: ':*:_dw:' \n/
                      liaf
              }
              output / All done! \n/
      }
```

The results:

```
      # crm window_6.crm
       Here's a word: 'This '
       Here's a word: 'is '
       Here's a word: 'the '
       Here's a word: 'Bash '
       Here's a word: 'FAQ, '
       Here's a word: 'version '
       Here's a word: '3.27, '
       Here's a word: 'for '
       .... and so on
```

This is as we would have expected. We can also WINDOW by lines, that is, looking for **`\n`** as a line separator instead of space as a word separator.

```
      {
              window   #  this WINDOW turns off the default read-to-EOF
              input (:bash_words:) [/usr/share/doc/bash-3.0/FAQ]
              {
                      window  /.*/   /\n/  (:_dw:) (:bash_words:)
                      match /./
                      output / Here's a line: ':*:_dw:' \n/
```

```
                    liaf
            }
            output / All done! \n/
    }
```
and get the following results:

```
    # crm window_7.crm
     Here's a line: 'This is the Bash FAQ, version 3.27, for Bash
    version 3.0.
    '
     Here's a line: '
    '
     Here's a line: 'This document contains a set of frequently-asked
    questions concerning
    '
     Here's a line: 'Bash, the GNU Bourne-Again Shell.  Bash is a
    freely-available command
```
... and so on

Notice that the newlines (including newlines that followed other newlines) stayed in place; they are still in the data window and so are still part of the output. We can also do this with the Linux spell check dictionary, in `/usr/share/dict/linux.words` . Note that the words in this list are already separated by newlines .

```
    {
          window    #  this WINDOW turns off the default read-to-EOF
          input (:dict_words:) [/usr/share/dict/linux.words]
          {
                window  /.*/   /\n/  (:_dw:) (:dict_words:)
                match /./
                output / Here's a word: ':*:_dw:' \n/
                liaf
          }
          output / `All done! \n/
    }
```

After working through the '-suffixes, and the -appends, we finally get some words. Here's a sample of it (the whole thing is about 5 megabytes long; we won't print it all out here)

```
    Here's a word: 'A
    '
     Here's a word: 'A&M
    '
     Here's a word: 'A&P
    '
```

```
 Here's a word: 'A'asia
'
 Here's a word: 'A's
'
 Here's a word: 'A-1
'
 Here's a word: 'A-OK
'
 Here's a word: 'A-and-R
'
 Here's a word: 'A-axes
'
```

If you run the code you will see that it seems to run very slowly- the issue is that every time we WINDOW with **`:dict_words:`** as a source, we have to cut the characters from the front of it- and that means shuffling 5 megabytes. There are better methods; the reader is encouraged to read on in the “tricks” section of this book.

Altering strings with ALTER, EVAL, TRANSLATE, and HASH

ALTER, EVAL, TRANSLATE, and HASH are the four “surgical” variable-modifying statements in CRM114. It's not too difficult to remember which does what - ALTER does a simple ALTERation, EVAL does a recursive evaluation (mnemonic hint: “recursiVE EValuation” - note the palindromic VE EV), TRANSLATE uses table-based translation to change the characters themselves (very much like the `bash` **`tr()`** command), and HASH takes the HASH of a string and turns it into a hexadecimal value. We've touched on ALTER and EVAL before; now we'll get deep into the differences.

Single-pass partial evaluation with ALTER

ALTER does **`\`**-constant substitution and **`:*`** variable substitution, and **`:+`** variable indirection only. It does only a single pass of evaluation (no recursive evaluation), and it doesn't evaluate **`:#`** string-lengths and **`:@`** math expressions. The good news is that this is intentional- you can use ALTER to build a string whose final value contains **`\`** and **`:*`** and **`:+`** and such, and the string will remain in that state. You can even **`:*`** that variable into another string, and because only one pass of substitution is done, the **`:*:`** strings will remain undamaged.

The syntax of ALTER is simple:

`alter (`*`:var_to_alter:`*`) / `*`var-expanded string (new var value)`*` /`

Here's a simple example of ALTERing a string (in this case, a string that's a variable overlapping part of **`:_dw:`**). Here's the code:

```
{
      output / The data window is --> :*:_dw: \n/
            #  grab hold of everything between foo and bar
      match (:my_var:) /foo.*bar/
      alter (:my_var:) /pepperoni pizza!/
      output / The data window has changed to -->  :*:_dw: \n/
}
```

and the result is:

```
# crm  alter_1.crm
I want a foo zebra bar
 The data window is --> I want a foo zebra bar

 The data window has changed to -->  I want a pepperoni pizza!

#
```

Now, let's try it with some embedded `:*`, `:#`, and `:@` strings, to demonstrate that ALTER only makes a single pass, and doesn't evaluate `:#` and `:@`. (Please forgive the use of ISOLATE to create a non-overlapping variable easily; we'll give a full description soon. For now, just accept that ISOLATE is a way to create a variable with a value without any possibility of this new variable interfering or overlapping with any other variable). Here's the code:

```
{
      window
      isolate (:colon:) /:/
      isolate (:star:) /*/
      alter (:_dw:) / Built varname  as :*:colon::*:star::_dw: /
      output /colon-star did NOT var-expand: ':*:_dw:' \n/
}
```

which yields:

```
# crm alter_2.crm
colon-star did NOT var-expand: ' Built varname  as :*:_dw: '
#
```

showing that ALTER only does one pass of evaluation.

Full Recursive Evaluation with EVAL

EVAL does `\`-substitution, then `:*` variable substitution, then `:+` variable indirection, then `:#` string-length calculation, and finally `:@` math evaluation – and repeats this process until one of three things happens:

- it stops when the string ceases to change
- it stops when the string re-visits a previous state
- it stops when too many evaluations have been done and the string hasn't stabilized (currently the default limit is 4096 iterations)

That's a big difference. You can play games with EVALs rules, coming up with a variable set that, when EVALed, turns either into something interesting (a palindrome, a musical score) or runs almost but not quite all the way to the evaluation limit. (puzzle- how would you write `factorial` using recursion in a single variable, that, when evaluated, generates the right answer?)

The syntax of EVAL is the same as ALTER:

```
eval (:var_for_result:) /input string for recursive evaluation /
```

Here's an example of EVALuation in action. We'll build a string and then print it in the pre- and post-EVAL forms. (and again, please forgive the use of ISOLATE). We also create a variable that we call **`:colonstar:`** to be the textural representation of `:*` . Here's the code

```
{
        window
        isolate (:colonstar:) /:*/
        isolate (:a:) /:*:colonstar::b:/
        isolate (:b:) /:*:colonstar::c:/
        isolate (:c:) /Triple nesting! You hit the jackpot!/
        output /The “standard expansion” of :a: is ':*:a:' \n/
        eval (:_dw:) /:*:a:/
        output /And the full recursive eval of :*:a: is ':*:_dw:'\n/
}
```

which gives the output:

```
# crm eval_1.crm
The “standard expansion” of :a: is ':*:b:'
And the full eval of :*:b: is 'Triple nesting! You hit the jackpot!'
#
```

Which looks pretty much right – but wait -- the **`:*:a:`** in the final output statement gets turned into **`:*:b:`** ! That's because OUTPUT itself does a basic var-expansion (no **`:#`**, no **`:@`**, no recursion) and so expanded **`:*:a:`** into **`:*:b:`**.

To actually get the output we want, we either need to fake out the `:*` in the final output statement, or use some other similar machination to prevent **`:*:a:`** from being expanded. Here's the code, using **`:colonstar:`** again to fake out the expansion. The **`:colonstar:`** variable evaluates to the string **`:*`**, but since OUTPUT only does a single pass of **`\`**, **`:*:`**, and **`:+:`** expansion, the `:*` stays in the string.

```
{
        window
        isolate (:colonstar:) /:*/
```

```
            isolate (:a:) /:*:colonstar::b:/
            isolate (:b:) /:*:colonstar::c:/
            isolate (:c:) /Triple nesting! You hit the jackpot!/
            output /The “standard expansion” of :a: is ':*:a:' \n/
            eval (:_dw:) /:*:a:/
            output /the recursive eval of :*:colonstar::a: is ':*:_dw:'\n/
      }
```

and it yields:

```
      # crm eval_2.crm
      The “standard expansion” of :a: is ':*:b:'
      the recursive eval of :*:a: is 'Triple nesting! You hit the
      jackpot!'
      #
```

exactly as desired.

HASHing a text

HASH is similar to ALTER – it does an in-place change to its input string, but with a difference. HASH does a basic var-expansion (no **:#**, no **:@**, no recursion) of its **/pattern/,** and then takes a very fast 32-bit hash of the string value. The 32-bit result is turned into an 8 hexadecimal character string, and that string is ALTERed into the **(:paren_arg:)** of the command.

The syntax is the same as ALTER and EVAL:

```
      hash (:paren_arg:) /var-expanded-string goes here/
```

By default, HASH uses **:_dw:** for both input and output.

Note that the hash is very fast, but only 32 bits wide. Thus it's not cryptographically secure (you can find a string that hashes to any given hash value in a few minutes on any reasonable PC); use HASH when you need a quick confidence check. Don't use HASH to secure international arbitrage funds transfers; realistically HASH is about as secure as a gym locker with a rented padlock – don't leave your wallet in it.

A recommended use of HASH is to create a fingerprint of a program or dataset, so you can have some assurance that you have the right version of a piece of code or a config file. In this case, the weaknesses of HASH are overridden by the low threat level of random chance.

Here's a sample program from the command line:

```
# crm '-{ hash; output /:*:_dw: \n/}'
hit ctrl-D twice, no carriage return, so that the input is a null string.
00000000
# crm '-{ hash; output /:*:_dw: \n/}'
foo
hit ctrl-D twice
00C3F3B5
# crm '-{ hash; output /:*:_dw: \n/}'
one two three four five six seven eight nine ten
hit ctrl-D twice
16E8E09F
#
```

which is perfectly reasonable. Note also that the hash of the currently executing program is available as the variable **`:_pgm_hash:`**, which is handy for quality assurance and testing; if
someone claims that they "haven't changed your program", but your diagnostic header shows a different **`:_pgm_hash:`** than you expect, you know someone is lying to you.

On the other hand, HASH is no substitute for `md5sum` for long files; here we see that it's quite a bit slower on a long file:

```
# time md5sum < /usr/share/dict/words
1e74aa5e2fbd6d91f8a370307d5022a1  -

real    0m0.104s
user    0m0.065s
sys     0m0.029s
# time crm '-{ hash; output /:*:_dw: \n/}' < /usr/share/dict/words
28B20400

real    0m0.524s
user    0m0.355s
sys     0m0.070s
#
```

The advantage of HASH over MD5 is strictly that of convenience; for datasets less than a megabyte it's comparable, and it's "already there" if you have a need for it. If you need security, speed over large data sets, or interoperability with non-CRM114 systems, you should use MD5[14].

14 Very recent developments in cryptography indicate that even well-designed checksums like MD5 or SHA1 may be compromised by a sufficiently determined opponent. Check the current state of the art literature if this matters to you.

TRANSLATEing Characters

TRANSLATE is very much like the bash **`tr()`** command; instead of doing variable substitution like ALTER and EVAL, TRANSLATE does character-by-character substitution. TRANSLATE needs two character sets, the from-set and the to-set, and changes characters in the from-set to the corresponding characters in the to-set.

Like the `bash` **`tr()`** command, TRANSLATE can translate from the from-set to the to-set, squeeze runs of the same character down to a single character, or delete characters in the from-set if no to-set is given. Unlike the `bash` **`tr()`** command, CRM114's TRANSLATE can also do reverse stepping correctly, inverted ranges on the from-set, and literal character ranges.

The syntax of TRANSLATE for translation from one charset to another is:

```
translate <flags> (:result:) [:source:] /from_charset/ /to_charset/
```

where **`:source:`** is the text input, **`/from_charset/`** is a set of characters that TRANSLATE will act upon (characters not in the from_charset will be retained but ignored), **`/to_charset/`** is the set of characters that the characters in from_charset will be changed into (first character to first character, second character to second character, and so on), and **`:result:`** is the variable where the result of the TRANSLATE will be placed (overwriting whatever was there before).

If there is no **`to_charset`** specified, then TRANSLATE translates all members of the from-charset to nothing – that is, it *deletes* those characters from the input and closes up the gaps; this statement looks like:.

```
translate <flags> (:result:) [:source:] /from_charset/
```

Deletion is handy when large amounts of the input can be discarded immediately, because they're in the wrong character sets. Be careful – omitting the to_charset can delete huge chunks of the wanted data.

TRANSLATE can take two flags - **`<unique>`** and **`<literal>`** . The **`<unique>`** flag tells TRANSLATE to compress all runs of the same character in the from_charset appearing two or more times into a single unique copy of the character. The remaining single copies of the character are then translated into the corresponding single character in the to charset

(In this case, <unique> does not mean "only one copy in the document", it means "only one copy per run.)

The **<literal>** flag tells TRANSLATE that the from_charset and to_charset are to be taken "as written", without any var-expansion, range expansion, or other alteration.

As in the `bash` **`tr()`** command, if there are too many characters in the to_charset, the extra characters never get used.

Differently from `bash` **`tr()`**, if there aren't enough characters in the to_charset, CRM114's TRANSLATE will start over at the start of the to_charset and reuse all of the characters again in sequence, unlike `bash` **`tr()`** that repeats the last character ad infinitum. The reason for this change in behavior is that reusing the entire to_charset is more useful for checksumming (or numerology!) than just using the last character ad infinitum. If you need the old behavior, just specify "more than enough" copies of the final character in the to_charset.

For example, the NIST definition of "soundex"[15] to transform a name into a more easily indexable phonetic format (one more immune to spelling variation) is:

```
Soundex Coding Guide
1 = B,P,F,V
2 = C,S,G,J,K,Q,X,Z
3 = D,T
4 = L
5 = M,N
6 = R

The letters A,E,I,O,U,Y,H, and W are not coded.
```

Soundex drops the letters a, i, o, u, y, h, and w, and drops all repeated sounds. Standard Soundex doesn't code the first letter but leaves it "as written", hence the name "Michael" becomes M24. A common variant codes initial letters as well, and "Michael" then codes to 524.

To perform a Soundex all-characters variant translation in CRM114, we can first drop all of the non-coded letters with one TRANSLATE, and then transform to the coding numbers with another TRANSLATE. Of course, we could interleave the letters and codes, and go slightly faster, but in the interest of ease of writing and debugging (and pedagogy) we'll write it as seven sequential translates- one to drop the non-coded letters and another to transform each of the coded ones. Here's the code:

15 From **`http://physics.nist.gov/cuu/Reference/soundex.html`**

```
#!/usr/bin/crm114
{
        translate /aeiouyhwAEIOUYHW/
        translate <unique> /bpfvBPFV/ /1/
        translate <unique> /csgjkqxzCSGJKQXZ/ /2/
        translate <unique> /dtDT/ /3/
        translate <unique> /lL/ /4/
        translate <unique> /mnMN/ /5/
        translate <unique> /rR/ /6/
        accept
}
```

We can run this code against some standard text:

```
# crm soundex.crm
the quick brown fox jumped over the lazy dogs back
3 222 165 12 2513 16 3 42 322 122
#
```

We see that this seems to work, except that there are duplicates in the “2” sound. The NIST webpage isn’t quite clear on whether “duplicate sound” means duplicated letters or duplicated soundex codes; we can produce the latter by adding one more TRANSLATE <unique> line:

```
#!/usr/bin/crm114
{
        translate /aeiouyhwAEIOUYHW/
        translate <unique> /bpfvBPFV/ /1/
        translate <unique> /csgjkqxzCSGJKQXZ/ /2/
        translate <unique> /dtDT/ /3/
        translate <unique> /lL/ /4/
        translate <unique> /mnMN/ /5/
        translate <unique> /rR/ /6/
        translate <unique> /123456/ /123456/
        accept
}
```

which gives us:

```
# crm soundex2.crm
the quick brown fox jumped over the lazy dogs back
3 2 165 12 2513 16 3 42 32 12
#
```

which may or may not be more accurate – the NIST web page doesn't quite give enough information to be sure.

Summary of Translate Charset Construction

The from char set and to char set are not regexes, nor are they simple strings (unless the **`<literal>`** flag is present.). Instead, they represent a shorthand notation to make it easy to write entire groups of characters. Here are the rules for creating a charset; we'll go through them in detail below:

- If **`<literal>`**, all characters stand exactly for themselves and the char set used is exactly as written. Not even "**`\`**" or "**`:*`**" is respected (which means embedded control characters are OK, but not characters that are escaped like **`\n`** or **`\xFF`** as there's no var-expansion whatsoever).
- Otherwise, normal var-expansion is done. During var-expansion, all of the **`\`**-escapes like **`\n`**. **`\t`**, etc are available, as are the hexadecimal characters expressed as **`\xFF`** and basic var-expansions like **`:*:foo:`** and **`:+:bar:`** .
- After var-expansion, range-expansion occurs:
  - A "**`-`**" (hyphen) character anywhere except as the first character or last character indicates a range; ranges are described below. To use a literal hyphen, use it as the first or last character.
  - A "^" (caret) character as the first character indicates char set inversion; inversion is described below. To use a literal caret, put it anywhere except the first character.
  - A caret as the first character <u>and</u> a hyphen as the second character still means inversion, but the hyphen is still treated as an ordinary character.
  - A caret as the second character is a normal character, and so ^^ creates the charset containing all characters except a carat.
  - A charset of three characters with a carat as the first, second, and third character (written ^^^) creates the char set containing only a single caret. (mnemonic: this is the NOT of (NOT of "not")).

Ranges in the Character Sets

The concept of a *range* is that instead of writing all of the sequential character byte-values out, we can just specify the endpoints, and let CRM114 fill in the middle.

For example instead of writing all of the letters from a to z:

```
abcdefghijklmnopqrstuvwxyz
```

we can just write:

```
        a-z
```

and get the same result. With CRM114's TRANSLATE, we can also specify a "reversed" range (this isn't permitted with the `bash` **`tr()`** command) such as:

```
        z-a
```

This expands to:

```
        zyxwvutsrqponmlkjihgfedcba
```

which might be useful for something. Remember that if you specify **`<literal>`** this will not happen – and if you don't specify **`<literal>`**, a hyphen anywhere except the first and last character will be used as a range creator, using the character to the left and the character to the right as the endpoints of the range.

It is OK for ranges of characters to overlap; the result is that the character appears more than once in the char set. For example:

```
        a-eb-f
```

yields the charset

```
        abcdebcdef
```

If this happens in the **`from`** char set, the last occurrence of a character is the one that corresponds to the character in the **`to`** char set (the previous ones are disregarded). If this is in the to char set, it means that more than one input character will result in the same output character. Neither situation is an error condition.

A useful example of ranges is the classic Usenet **`rot13`** "cypher", used to hide the punch line of jokes. The **`rot13`** cypher is just "rotate through the alphabet by 13 characters", thus **`a`** becomes **`n`**, **`b`** becomes **`o`**, and so forth to **`m`** becoming **`z`**. The letter after **`m`** (that is, **`n`**) would become the letter after **`z`**, but since there is no letter after **`z`**, we loop around and so an **`n`** becomes the letter **`a`** instead. Because of this loop-around symmetry (and because the English alphabet has 26 = 2 * 13 characters) the **`rot13`** code is it's own inverse; to decode **`rot13`**, just encode again.

Here's an example of **`rot13`** in action. We will take the defaults of reading input from stdin, using :_dw: for both the data source and result destination for the TRANSLATE statement. The crm114 program source is:

```
#!/usr/bin/crm
translate /a-z/ /n-za-m/
accept
```

Note that this program only specifies the 26 lowercase letters; uppercase letters, spacing, and punctuation won't be altered. The results are fairly unintelligible and suitably secure for a joke punchline.

```
# crm rot13_1.crm
The Quick Brown Fox Jumped Over The Lazy Dog's Back.
hit return, then ^D
Tur Qhvpx Bebja Fbk Jhzcrq Oire Tur Lnml Dbt'f Bnpx.
#
```

We can demonstrate that this **rot13** encoding is it's own inverse, by feeding the program it's own output:

```
# crm rot13_1.crm
Tur Qhvpx Bebja Fbk Jhzcrq Oire Tur Lnml Dbt'f Bnpx.
hit return, then ^D
The Quick Brown Fox Jumped Over The Lazy Dog's Back.
#
```

Standard USENET **rot13** includes rotation of the capital letters as well as the lowercase letters; we can do this by specifying the uppercase ranges as well in the char sets:

```
#!/usr/bin/crm
translate /a-zA-Z/ /n-za-mN-ZA-M/
accept
```

which gives us the "traditional" rot13:

Besides performing **rot13** cyphers, TRANSLATE ranges are quite useful for "block conversion" of control characters to spaces (or nothing), and to alter non-ASCII text found in other character sets into moderately reasonable 7-bit ASCII. For example, the ISO8859-* (European languages) font set and KOI8-R font set (Russian) can be roughly transliterated to pronounceable ASCII.

Note that (sadly) such translations don't work well on multibyte characters nor Unicode, nor does it actually translate words, only 8-bit characters. There are external programs that might help in that case; see the section on Asian Languages for a few hints.

Inversion in the From_Charset and To_Charset

Sometimes it's easier to decide what characters you don't want to manipulate, rather than the characters that you do. In this case, you can use range inversion to change one or both char sets from every character in the char set to every character not in the char set.

To indicate inversion of a char set, add a caret ^ as the first character of the input string. This caret isn't used as part of the char set string; instead, it signifies that after all other expansions are finished, that the char set is to be inverted. The caret must be the first character of the input string; anywhere else and it's just a caret, not an inversion operator.

The inverted char set "loses" the sequence of characters that the original char set had; inverted char sets are always sorted, starting at **`\x00`**, and working upwards to **`\xFF`**, assuming those characters are in the char set. Even if the original char set had duplicated characters, the inverted char set has either 0 or 1 occurrence of any particular character.

Here's an example- suppose we wanted to remove all of the characters from a file except the digits 0 through 9. Rather than hand-typing all of those characters, we can use carat to get the inverted charset of the numbers 0 through 9:

```
^0-9
```

which yields (please forgive the **`\x`** notation):

```
\x00-\x2F\x3A-\xFF
```

(the **`\x2F`** is an ASCII forward-slash, and the **`\x3A`** is an ASCII period)

Inverted char sets are most commonly used in deletion mode (where only the from_charset is specified, but no to_charset)

For example, assume the task is to remove all text except what might be phone numbers from a text. If we only removed a-z and A-Z, that would still leave punctuation. If we hand-code the punctuation, that still leaves the control characters and arbitrarily large amounts of whitespace. This makes a simple-sounding problem a lot more complex (and bug-laden!) than it needs to be.

We might decide to do this as a three-step process:

1. Get rid of any "obviously not useful" character (leaving only phone numbers and

whitespace)
2. Turn all whitespace characters into spaces
3. Get rid of excess spaces.

We'll accept that phone numbers might be written in the American style with hyphens like (123)456-7890 or in the European style with dots like (01)2345.67890 .

It's quite true that we could do this all with just MATCH regexes followed by ALTERs, but it's much faster (and easier to understand) this version using TRANSLATE.

Here's the code:

```
#!/usr/bin/crm
# get rid of everything that isn't a number, whitespace, or a punct.
translate /^-0-9.() \n\t\v/
#                turn all the whitespace into simple spaces
translate /\n\t\v/  / /
#                                  compress excess spaces away
translate <unique> / / / /
#                                         all done!
accept
output /\n/
```

When run, we get:

```
# crm translate_phonenum.crm
this is a test
Give me a call at 1-800-555-1212 and ask for Bill
or in Ireland at (085)4525.282.1700 when I'm there
or just call 411 for information
 1-800-555-1212 (085)4525.282.1700 411
#
```

(note that we had to add a final **\n** newline, as the program above deleted all newlines in the data window.)

In summary, use TRANSLATE in preference to a MATCH/ALTER loop if you are altering single characters, deleting single characters, or compressing runs of the same character to a single copy. It runs faster and is easier to understand than the MATCH/ALTER loop.

Making non-overlapping strings with ISOLATE

In the previous discussions, we've often talked about a “non-overlapping variable” without describing how that was different than a typical MATCH-created variable; we've occasionally even said “an ISOLATEd variable”. Now we'll find out what this really means.

To ISOLATE a variable in CRM114 means to make a copy of the variable's data string such that it does not overlap with any other variable in the entire program. As of the execution of the ISOLATE statement, this guarantee is absolute. Of course, nothing stops you from using MATCH to create another overlapping value in the next statement, so remember that an ISOLATEd variable is only non-overlapping until you make it otherwise.

Syntactically, ISOLATE is simple; there are two forms, one of which has an optional initializer value. Here's the syntax:

```
isolate (:my_var:) <flags> /optional var-expanded initial value/

isolate (:this_var: :that:var: :another_var:)
```

In the first form, the optional initial value is allowed; if there is no initializer given, the previous value of the variable is used (if there was one). If there was no previous value for the variable, then the null string (a string of length zero) is used as the initializer value.

In the second form, multiple values are all ISOLATEd, but no initial value is supplied. This is the recommended way to use command-line value settings, by ISOLATEing each possible command line value; values that weren't set in the command line are thereby initialized with a null string, and values that were set in the command-line keep their command-line settings.

The one flag allowed in ISOLATE is **`<default>`**. The **`<default>`** flag is defined to mean “only do the ISOLATE if no previous value has been set for this variable”. This is designed to work with command-line arguments; if you don't supply an argument on the command line, the ISOLATE **`<default>`** statement will put the **`/var-expanded initial value/`** there, and if you do supply an argument, the ISOLATE **`<default>`** statement has no effect whatsoever. In either case, an ISOLATE is still an executable statement, so if you are going to use a WINDOW statement to prevent read-to-EOF then the WINDOW must go before the ISOLATE **`<default>`** statement.

You can use multiple ISOLATE **`<default>`** statements on the same variable – only the first one executed will have any effect. This allows for a particularly cute trick: have different default values set up for some program variables depending on the setting of other command-line-settable program variables. For example, your program can have a command-line argument named "**`--advanced_user`**". The program startup code can check the value of **`–advanced_user`** and if it's SET, then the program uses a series of ISOLATE **`<default>`** statements to turn on some special (or dangerous) features. Whenever the special **`--advanced_user`** isn't set, a different set of default values can be turned on. An intermediate user, who is afraid of **`–advanced_user`** but needs some particular parameter set can turn on only those parameters they need, leaving the more dangerous features disabled.

As you might expect, ISOLATEd variables can't share space with each other or the default data window **`:_dw:`** . In fact, the ISOLATEd variables, plus any string constants needed by the engine, plus the names of all variables, all inhabit a large preallocated pool of storage called the isolated data window. Like the default data window, the isolated data area is of fixed size, to prevent DoS attacks. The default size of this isolated data window is the same size as the default window; that is, 8 megabytes, and this size can be overridden with the -w flag.

If you like to live dangerously, you can peek at what's in the isolated data window by looking at the variable **`:_iso:`**. But be warned- writing into the isolated data window can break your program in horrible ways.

Once a variable has been ISOLATEd, you can use it just as before, and if you use it as the domain in a MATCH, you can create overlapped sections within it. These new sections are full-fledged variables as well.

An interesting point is what happens when an ISOLATEd variable is re-defined by using it as a matched output in a MATCH statement on **`:_dw:`** . In this case the MATCH undoes the ISOLATE; the variable is now overlapping part of the **`:_dw:`** and will change as **`:_dw:`** changes.

But... what if we MATCH a variable onto our ISOLATEd variable, and then un-ISOLATE the first variable? Does the second variable un-ISOLATE as well? Or does the second variable continue its existence on its own, as an ISOLATEd variable?

The answer is that it's the second variable that continues to be ISOLATEd as an independent variable, even though the original hosting variable is long gone. Here's example code showing this:

```
{
      isolate (:my_dw_copy:)  /:*:_dw:/
      match [:my_dw_copy: 0 5 ] /.*/ (:first_five:)
      output /My first-five characters are ':*:first_five:' \n/
      alter (:_dw:) /hello world/
      output /data window is now ':*:_dw:'  \n/
      output /my first-five chars are still ':*:first_five:' \n/
}
```

and gives the results:

```
# crm isolate_1.crm
abcde12345
return, then ctrl-D
My first-five characters are 'abcde'
data window is now 'hello world'
my first-five chars are still 'abcde'
#
```

which is exactly as we would expect. The ISOLATEd variable didn't change, even when the default data window **`:_dw:`** did.

CRM114 has a built-in, in-flight storage reclaimer. Whenever a string changes length, the storage reclaimer repacks the remaining strings so that there are no voids in the storage pools and the maximum allowed length for a new variable is always immediately available. This includes the situation when a variable was ISOLATEd and then ceases to be isolated; the reclaimer determines what previous string space segments of the variable are no longer part of any accessible variable and compresses them away, keeping all of the variable bookkeeping correct.

The net result is that CRM114 has completely automatic storage allocation and reclamation, and most programs will run efficiently without the designer taking any special care to maintain storage pools or preallocation zones.

The downside of this is that a long isolated variable that changes length repeatedly will cause repeated small motions of the data window- which are bytewise copying operations. This is slightly wasteful of CPU time. The fix (should you run into this performance problem) is to isolate the long variable and then create a second variable within that long variable. As long as you don't do any ALTERs to this second variable that change it's length, there will be no length change in the outer ISOLATEd variable and thus no bytewise copy operations which can speed up some programs.

Subroutine CALL, RETURN, and EXIT

This chapter discusses problem-state structure- that is, subroutine labeling, calling, returning, and overall program exit values. There are four pertinent types of statement here – a statement label (with subroutine extensions) , the CALL, the RETURN, and, eventually, program EXIT.

Because of CRM114's overlapped-string data structure, CALLed subroutines in CRM114 don't have their own address spaces. (we tried it that way, and it was awful; overlapping caused unpleasant, ugly aliasing in the namespace.) So, the current releases of CRM114 have a shared caller / callee memory model; call-specific data is passed down by a freshly-ISOLATEd variable, and the return value is similarly passed upward. This isn't perfect; any recursive subroutines need to be written knowing that by default they have no storage that isn't shared with every other copy of themselves up and down the call stack. Recursive routines that can share storage are called “tail-recursive” routines and it's a good way to write code in any case.

If your recursive routines don't easily fit into the tail-recursive model, it's possible to create recursively-labeled variables; this is appropriate for both recursion and modularity in library routines.

If you absolutely NEED ironclad partitioning of memory between caller and callee, see SYSCALL for a way to do this.

Because EXIT, CALL, routine labels, and RETURN are so intimately related, we'll describe all four first, and then give examples.

EXITing your program

Sometimes it's useful for a program to return an exit code; EXIT lets you do this. You may also want to terminate program execution without waiting to run “off the end” of a program. The EXIT statement allows this as well.

The EXIT statement has this syntax:

```
        exit /:optional_var_expanded_exit_value:/
```

The exit value should be the string representation of a small positive integer (from 0 to +127). If there is no exit value, or CRM114 can't turn the value into an integer, a 0 is used as the exit value.

For example:

```
        # crm '-{ window; exit /123/ }'
        # echo $?
        123
```

showing that you can indeed set your own exit codes. But watch out – some *NIX variants use a negative value to indicate pipe problems or SIGNAL values, and folds user-supplied negative values into the positive domain (that is, -128 to -1 get folded to values from 128 to 255). For example:

```
        # crm '-{ window; exit /225/ }'
        # echo $?
        225
        # crm '-{ window; exit /-31/ }'
        # echo $?
        225
        #
```

Meanwhile, on other *NIX variants, a SIGKILL (signal value of -9) will show up in `bash` as a positive number between 128 and 255. For example SIGKILL can get mapped to 137:

```
        # crm '-{ liaf }'
        .......  infinite loop ...... kill -9'ed from another window
        [1]+  Killed                    crm '-{ liaf }'
        Killed
        #  echo $?
        137
        #
```

Exit codes are particularly handy for writing text filters that need to talk to `procmail`; this is worth remembering.

Exit Code Starting Base Value

In some situations, it's necessary to change the exit code values that the CRM114 engine itself will use. Normally, CRM114 will exit with a 0 on successful program completion, and 1 on any fatal error, including system internal errors. Unfortunately, the user program may want to use the values 0 and 1 for its own status results and a “1” may not mean error to the caller, but rather be a legitimate non-error return.

The workaround for this is to reserve some exit return codes for the user program with the **`-E N`** command line flag.   If **`-E N`** is present (and with a nonzero **`N`**) then all exit status codes from 0 to N are reserved to the user program; all of CRM114's error return values will be moved to start at N+1 with one exception- a normal exit either because of an EXIT statement or by running off the end of the program returns an exit code of 0 whether or not **`-E N`** is used. Additionally, instead of sharing all fatal errors including system internal error onto exit code 1, each CRM114 internal error gets its own value. How many such error returns exist varies with the release, but at this writing, a user program may use all exit codes up to 100 and still allow each fatal, unrecoverable CRM114 exit error to have its own exit code as well.

Subroutine Labels

A subroutine label looks like any other statement label; it has only one option – the name of an optional variable. This variable is freshly ISOLATEd during the calling sequence and gets the var-expanded string passed downward by the CALL.

The syntax (with and without the optional transfer variable) is:

```
      :my_subroutine_label:
      :my_subroutine_label:  (:my_argument_transfer_variable:)
```

Note that there's no additional `{}`-bracketing required for a subroutine; a subroutine can begin anywhere, and ends when a RETURN statement executes or your program EXITs. There's no *implied return* at the end of a program; running off the end of a program is an EXIT with value 0, not a RETURN to a call.

The argument transfer variable is optional – if you include it in the subroutine label statement, and the caller supplies an argument string, the argument transfer variable is freshly ISOLATEd and initialized with the CALLers var-expanded argument string. If the caller does not supply an argument string, then the subroutine gets an empty string (zero length) as its argument transfer variable.

Subroutine CALL

The CALL statement requires the label to be called (in **`/slashes/`** ; the label is var-expanded, so it is possible to compute the name of the subroutine you want to call). The label to be called is required as there isn't really any “default subroutine”.

Optionally, you can send an arbitrary argument string to the called routine. Put this argument string in **`[boxes]`**; the routine will receive this as the optional argument transfer variable. The variables passed can be var-expanded, keyword-marked, or ; the difference is that if you pass a copy, the original won't be altered, but if you pass by name, the subroutine can alter the original easily.

You may also optionally specify a return value variable in **`(parens).`** If present, a freshly ISOLATEd variable will be created on return and whatever the subroutine RETURNed will be placed there. If you don't supply a return variable, no return value is accepted (and if the routine produces one, the string value is generated, then thrown away).

The overall syntax for CALL is:

```
      call /:routine_name:/ [:var_exp_arg_list:] (:return_value:)
```

RETURNing from a CALL

To return from a CALL, use the RETURN statement. Return has one optional argument, a string that is var-expanded and offered to the caller as a return value (if the caller doesn't specify a return value variable, the return value string is built anyway, then thrown away).

The syntax for a RETURN is

```
      return / var-expanded return string /
```

Call/Return Examples

To make it easier to understand how CALL and RETURN work, we'll have a few examples. Of course, the goal of these examples is not to do anything deep, but rather to show how to CALL and RETURN work in the context of CRM114 code.

CALL without Argument Transfer

Here's a short routine that takes a string, and replaces “foo” with “bar”. This example does not transfer arguments either downward to the subroutine nor back up to the caller; everything is assumed to be in fixed variable names.

Here's the code (including the calling sequence):

```
{
      output / This program replaces 'foo' with 'bar' \n /
      output / Initial text= :*:_dw: \n/
      call /:replace_it:/
      output / New text= :*:_dw: \n/
      exit
}
:replace_it:
{
      match (:my_foo:) <fromend> /foo/
      alter (:my_foo:) /bar/
      liaf
}
return
```

giving the unexciting:

```
# crm call_1.crm
lion tiger foo snake
milkshake foo cable TV
return, then ctrl-D
 This program replaces 'foo' with 'bar'
  Initial text= lion tiger foo snake
milkshake foo cable TV

 New text= lion tiger bar snake
milkshake bar cable TV

#
```

CALL with Argument Transfer Downward

There's not much call for replacing “foo” with “bar”, so let's use argument passing to tell our subroutine what we want to replace with what. Remember, **`:_arg2:`** is the first of the “user” arguments (**`:_arg0:`** is the name of the CRM114 engine, and **`:_arg1:`** is your program's name).
In this case, we need to transfer two arguments downward only (from caller to callee); the return value is still in a fixed variable (in this case, in **`:_dw:`**)

Here's example code:

```
{
      output /We replace ':*:_arg2:' with ':*:_arg3:' \n /
      output / Initial text= :*:_dw: \n/
      call /:replace_it:/ [:*:_arg2: :*:_arg3:]
      output / New text= :*:_dw: \n/
      exit
}
:replace_it: (:my_args:)
match [:my_args:] /([[:graph:]]+) ([[:graph:]]+)/  \
      (:: :from_string:  :to_string:)
output / Replacing ':*:from_string:' with ':*:to_string:' \n/
{
      match (:my_foo:) <fromend> /:*:from_string:/
      alter (:my_foo:) /:*:to_string:/
      liaf
}
return
```

This gives:

```
# crm call_2.crm chocolate vanilla
I would like a chocolate milkshake
with chocolate sprinkles on top.
return, then ctrl-D
We replace 'chocolate' with 'vanilla'
  Initial text= I would like a chocolate milkshake
with chocolate sprinkles on top.

 Replacing 'chocolate' with 'vanilla'
 New text= I would like a vanilla milkshake
with vanilla sprinkles on top.

#
```

Thus, we see how to pass arguments. There are a few things to watch out for- because there is only ONE argument received by the routine, you have to exercise caution in how you assemble the argument and how you MATCH to extract the arguments.

CALL and RETURN are a tricky part of CRM114. The problem is that the overlapping-string data model (which implies a large amount of side-effecting) is simply incompatible with the commonly accepted notion of a subroutine having independent variables. The ultimate solution to this problem (yielding independent namespaces for each subroutine) is to be found in SYSCALL, below.

Another (smaller) issue is that argument transfer through the single transfer variable seems arcane to the uninitiated; in reality it means that every CRM114 subroutine implements a vararg-type interface by default.

CALL with Argument Transfer Both Down and Up

In this example, we'll do a CALL that transfers arguments both downward and back upward. The function is still the same – replacement of a string, but we'll pass three arguments downward – the string to be altered, the "replace-from" argument, and the "replace-to" argument.

Here's the code passing our arguments by value:

```
{
      output /We replace ':*:_arg2:' with ':*:_arg3:' \n /
      output / Initial text= :*:_dw: \n/
      isolate (:out_value:)
      call /:replace_it:/ \
            [ :*:_arg2: :*:_arg3: :*:_dw:] \
            (:out_value:)
      output / New text= :*:out_value: \n/
      exit
}
:replace_it: (:my_args:)
match [:my_args:] / ([[:graph:]]+) ([[:graph:]]+) (.*)/  \
      (:: :from_string:  :to_string: :retval:)
output / Replacing ':*:from_string:' with ':*:to_string:' \n/
{
      match [:retval:] (:my_foo:) <fromend> /:*:from_string:/
      alter (:my_foo:) /:*:to_string:/
      liaf
}
return /:*:retval:/
```

which gives the expected result:

```
# crm call_3.crm red blue
I like the red paint on the new Chevys.
I think the red looks fast.
return then ctrl-D
We replace 'red' with 'blue'
  Initial text= I like the red paint on the new Chevys.
```

```
I think the red looks fast.

 Replacing 'red' with 'blue'
 New text= I like the blue paint on the new Chevys.
I think the blue looks fast.

#
```

Again, as we expected. However, a word to the wise here – note that the first two arguments to our subroutine are transferred "literally", that is, they appear as single words in the argument string of the subroutine. The third argument (the text to be translated) also appears as simple text – it's all the remaining text. This is sometimes called a *pass by value* in computer language circles, because we didn't pass the variable, we passed only a copy of the value (and, in this case, ran all of the different arguments together with only spaces to separate them). The good news is that the passed value is a copy, and the original values are safe from damage by the subroutine.

In this simple example this is not a limitation. However, in situations where more than one argument may have embedded spaces, this simplistic argument transfer will simply not work. We want a way to pass not just the value of a variable but rather the *name* of the variable to the subroutine. This is called *pass by name* and we'll see how to do that next.

CALL with Arguments Passed By Name

In a pass by name CALL, we don't use the **:*** var-substitution operator in the CALL statement. Instead, we give the name of the variable right to the subroutine and let the subroutine resolve what the variable means. This usually is done with the **:+** (colon-plus) var-indirection operator.

Var-indirection is very similar to **:*** var-expand operator but performed twice in immediate succession, instead of just once. The reason for this is that any MATCH in the subroutine that tries to take apart the incoming argument string containing pass by name arguments will get variable names, not variable values. For example, if we did a

```
call /:my_routine:/ [ :my_arg_1:  :my_arg_2: ]
```

and then used a MATCH to take apart the argument transfer string as in:

```
:my_routine: (:my_arg_string:)
match / ([[:graph:]]+) ([[:graph:]]+) /   (:: :a1: :a2: )
```

then the values of `:*:a1:` and `:*:a2:` would be the strings `":my_arg_1:"` and `":my_arg2:"` rather than the values of those two variables. Now, the string values `":my_arg_1:"` and `":my_arg_2:"` would be fine for being the result of an ALTER, but we can't read what's in the variables described by the names `:my_arg_1:` and `:my_arg_2:` .

To get around this "the variable name is not the variable value" problem, it's possible to arrange a nested set of variables that var-expand into a repeated `:*` and then subject that to an EVAL, but there's a much easier way in this common case: use the `:+` var-indirection operator.

Using `:+` var-indirection means "this var contains the name of a var that contains what I want. Go get me what I want". Typically, you use `:+` when a varname was passed in a subroutine call and you need to access the contents of the variable whose name was passed. To use variable restriction or do an ALTER the var whose name was passed, just use `:*` on that variable – the `:*` will var-expand to the variable name, exactly as you want for a restriction or ALTER target.

Here's our example, rewritten to use pass by name. Note that the advantage of this is that any of the subroutine arguments can now contain embedded spaces with no danger of misaligning what argument is what. The downside is that because we are now passing the name of the variable, the subroutine can ALTER the variable itself (the variable is no longer safe!)

Here's the code:

```
{
        output /We replace ':*:_arg2:' with ':*:_arg3:' \n /
        output / Initial text= :*:_dw: \n/
        isolate (:out_value:)
        call /:replace_it:/ \
                [ :_arg2: :_arg3: :_dw: ] \
                (:out_value:)
        output / New text= :*:out_value: \n/
        exit
}
:replace_it: (:my_args:)
output / Argument string received: -->:*:my_args:<--\n/
match [:my_args:] /([[:graph:]]+) ([[:graph:]]+) ([[:graph:]]+)/  \
        (:: :from: :to: :text:)
output / Replacing ':+:from:' with ':+:to:' \n/
{
        match [:*:text:] (:my_foo:) <fromend> /:+:from:/
        alter (:my_foo:) /:+:to:/
        liaf
}
return /:+:text:/
```

This example works just as well as the previous one:

```
# crm call_4.crm red blue
foo bar red foo bar green
lots and lots of red ink
```
return, then ctrl-D
```
We replace 'red' with 'blue'
  Initial text= foo bar red foo bar green
lots and lots of red ink

 Argument string received: --> :_arg2: :_arg3: :_dw: <--
 Replacing 'red' with 'blue'
 New text= foo bar blue foo bar green
lots and lots of blue ink

#
```

but the advantage of this new pass by name is that we can have arguments with embedded spaces that don't get all mixed up. Here's an example (note we use single quotes to force `bash's` command line parser to not break our argument strings at spaces).

```
# crm call_4.crm 'The Republican' 'The Democrat'
The Chair recognizes The Republican from Kansas.
```
return, then ctrl-D
```
We replace 'The Republican' with 'The Democrat'
  Initial text= The Chair recognizes The Republican from Kansas.

 Argument string received: --> :_arg2: :_arg3: :_dw: <--
 Replacing 'The Republican' with 'The Democrat'
 New text= The Chair recognizes The Democrat from Kansas.

#
```

If you try this with the previous pass by value example, you'll find that the argument string MATCH statement will take the wrong arguments – the **`:from:`** string will be "The" and the **`:to:`** string will be "Republican".

CALL with Recursion

In the previous examples, we never allowed a recursive call. This is because CRM114's standard CALL model doesn't separate the variable name spaces of caller and callee, so there's no way to keep a local variable without some extra work on the part of the programmer.

The solution to this issue is to use **`:_cd:`** (the Call Depth system variable) to generate differing names for the variables in each recursive call level. The **`:_cd:`** variable gives the current call depth; the top level program is at call depth 0, the first callee is at call depth 1, and so on. This is particularly useful when using the indirection operator **`:+`** as the expression **`:+:foo_:*:_cd::`** will fetch the value of variable **`:foo_0:`** if at toplevel, **`:foo_1:`** in the first callee, **`:foo_2:`** in the second callee, and so forth. Although there's no requirement nor enforcement of using the form **`varname_call-level`** , it's a good convention to adopt (and those who follow you will curse you if you violate this rule).

Here's the classic example of a recursive program to evaluate the factorial function, taking the first user argument (**`:_arg2:)`** to be the number to be factorialized. The variables **`in_calllevel, out_calllevel`**, and **`child_calllevel`** are all recursively protected by appending the call level **`:_cd:`**. Because the variable **`:nm1:`** (n minus 1) is used only for the status PRINT and then in the call to the subsidiary caller, and not used thereafter, we don't have to use **`:_cd:`** recursive protection on **`:nm1: .`**

```
window
isolate (:res:)
call /:fac:/ [:_arg2:] (:res:)
exit
:fac: (:in_:*:_cd::)
{
      output / call level :*:_cd: input var name :in_:*:_cd:: value
is ':+:in_:*:_cd::' \n/
      {
            match [:in_:*:_cd::] / 1 /
            output / Input is 1, so return value is 1 \n/
            return /1/
      }
      isolate (:out_:*:_cd:: :child_:*:_cd:: :nm1:)
      eval (:nm1:) / :@: :+:in_:*:_cd:: - 1 : /
      output / Recursing with arg :*:nm1:  \n/
      call /:fac:/ [:*:nm1:] (:child_:*:_cd::)
      output / Level :*:_cd: received ':+:child_:*:_cd::' \n/
      eval (:out_:*:_cd::) /:@: :+:in_:*:_cd:: * :+:child_:*:_cd:::/
      output / Returning with ':+:out_:*:_cd::' \n/
      return /:+:out_:*:_cd::/
}
```

The result shows the recursive nature of our code:

```
 # crm call_5.crm 5
 call level 1 input var name :in_1: value is '5'
 Recursing with arg  4
 call level 2 input var name :in_2: value is ' 4 '
 Recursing with arg  3
 call level 3 input var name :in_3: value is ' 3 '
 Recursing with arg  2
 call level 4 input var name :in_4: value is ' 2 '
 Recursing with arg  1
 call level 5 input var name :in_5: value is ' 1 '
 Input is 1, so return value is 1
 Level 4 received '1'
 Returning with '2'
 Level 3 received '2'
 Returning with '6'
 Level 2 received '6'
 Returning with '24'
 Level 1 received '24'
 Returning with '120'
#
```

The call levels step inward, as we calculate successively smaller factorials, and then step outward again, as each callee returns with the factorial of the number they were called with.

This is not the only way to write a recursive program; if one uses variables carefully, it's possible to design factorial and many other recursive algorithms such that no variable is used across a recursive CALL / RETURN pair, and so the reuse of that variable inside of the callee recursive routine does not break the caller's use of the same variable. This is an extension to the concept of tail recursion; it's very pretty if you can manage it, but **`:_cd:`** protection is not computationally expensive so it's not a requirement that your CRM114 subroutines be tail recursive.

Deciding when to use pass by name, pass by value, pass by recursion, or pass by agreeing what variable does what is an exercise in good software engineering. There are no hard-and-fast rules here.

If you have a segment of code that repeats a few times, and always on the same variables, then pass by agreeing is perfectly reasonable (and *very* fast). On the other hand, pass by name will give you the maximum reusability of code and minimum chance that you will have problems with deconstructing the argument transfer string at the cost of slightly slower execution speed. Pass by value is in the middle; it allows more code reuse, but you still can have arguments that “accidentally” use the same delimiter as your argument transfer deconstruction, in which case your program is in a world of hurt and you'll spend a lot of time answering bug reports.

External Process control with SYSCALL

CALL and RETURN deal only in a local, memory-sharing routines, which can be inconvenient at times. The SYSCALL statement is a generic way for a CRM114 process to fork, either to allow a computation in a duplicate copy of its own address space, or to call any executable program in the system. For example, you can SYSCALL to `lynx` to fetch web pages, to `wget` to move files around the network, and to `netcat` (also known as `nc`) to establish a TCP connection to an arbitrary port on an arbitrary machine.

The SYSCALL statement is actually a single-statement interface to `fork()` with added synchronization and pipe augmentation. In SYSCALL, the forked child process is termed the *minion* process; this is either a copy of the current CRM114 program or a `bash` command. CRM114 does not enforce any anti-DoS limit on the number of minion processes (because it would be trivial to write code to do binary-tree forking, thereby circumventing any limit greater than 0)

The syntax of SYSCALL is:

```
      syscall /command or label/ (:input:) (:output:) (:status:) <flags>
```

Note that this is one of the few places in CRM114 where multiple parenthesized arguments are used. The language designers apologize for the inconvenience.

Since SYSCALL is a "call", you must supply a `bash` command to run, or a CRM114 **`:label:`** to branch to. This command or label will be the first thing that happens in your freshly forked minion process.

If the command is a `bash` command, the forked minion process will only share the environment variables and **`stderr`** with the CRM114 process; if the command is a **`:label:`** in the current program, the minion will also have a complete copy of the current data window **`:_dw:`** and a complete copy of all of the ISOLATEd variables. These are `fork()` copies, so changes the minion makes to these variables are not seen by the main CRM114 program.

As CRM114 never holds an open file, there are no open files available to passed or visible in the minion process[16]. Only **`stderr`** is shared with between the master CRM114 process and the minion (or minions) and it's shared across all of the minions of the current process.

The first parenthesized variable, *`input`*, is a string that is var-expanded and passed to the minion via `stdin`. It can have multiple variables; the full string is built, var-expanded, then passed to the minion (there is an anti-DoS limit on the maximum block size to move in a single SYSCALL; the default is 2 megabytes. If you need to move more, see **`<keep>`** below)

The second variable, *`output`*, is a variable that will receive the output of the minion. Var-expansion is done, and the first variable name found in the var-expansion gets the `stdout` result of the minion. Again, an anti-DoS blocksize limit exists; the default is also 2 megabytes.

The third variable, *`status`*, is a variable that is filled in with useful information about the minion, such as the PID, the file descriptors of the pipes that were used, etc. When the minion finally exits, the exit code is written into the status variable. While the minion is alive, the status variable will look something like this:

```
      MINION PROC PID: 10373 from-pipe: 6 to-pipe: 5
```

and when it exits and the exit code is retrieved, it changes to something like:

```
      DEAD MINION, EXIT CODE: 123
```

Of course, the minion process PID and exit code will vary to match your situation. There are a few conditions (such as *asynchronous minions)* in which CRM114 won't have updated the status variable; in those cases a zombie process will remain until another SYSCALL occurs; SYSCALLing to a label that EXITs immediately will clear all asynchronous zombies (but note that the status variables are not updated for asynchronous minions; after all, they are disconnected).

The three variables (input, output, and status) all are optional. If you want an output variable, you must supply an input, which may be empty, such as **`()`** . If you want the current status or exit code, you must supply both the input and output variables, both of which may be empty; **`()()`** is acceptable.

16 Actually, this is a comforting lie. The pipes leading to stdin and stdout of the minion *are* held open, and as they are pipes, they are shared with subsequent minions. So, it's not entirely true that there is no such thing as open files in CRM114, they're just well-hidden.

SYSCALL flags

SYSCALL also accepts two flags: **`<keep>`** and **`<async>`**. These flags are mutually exclusive. They control how CRM114 deals with the minion process.

By default, a CRM114 minion is like a CRM114 file I/O – never held open, always freshly created, used once, then thrown away (more specifically, the minion pipes for input and output are set up, the minion is forked, the minion input is piped to the minion, followed by EOF, CRM114 waits for the minion to complete, reads the minion's output pipe (again with an anti-DoS limit of 2 megabytes), the minion is `kill()`ed, and finally `wait()`ed for to obtain the final exit status and prevent zombie processes from accumulating..

The **`<keep>`** flag tells CRM114 to keep the minion around; all of the essential information about the minion is stored as human-readable text in the **`status`** variable. Instead of sending the minion the input string and then EOFing the pipe to minion `stdin`, it keeps the pipe open and available for more data. This allows CRM114 to operate most conversational-mode programs, like `ftp, bc, units, nc, telnet, lynx,` and `ssh`. As long as every SYSCALL to a minion process has the **`<keep>`** flag, the minion's I/O pipes will continue to be kept open, and operating in lockstep with the master CRM114 program. To finally exit the minion program, just do any SYSCALL without the **`<keep>`** flag (using a null **`()`** input argument is fine); that closes the minion's pipes and lets the minion gracefully exit on EOF.

The **`<async>`** flag is the reverse of **`<keep>`**. An **`<async>`** process is forked, sent its input, EOFed, a short attempt is made to read any return, and then the main CRM114 program continues onward. The **`<async>`** minion process will see all reads past the passed input on `stdin` as EOF; anything the minion sends to `stdout` is discarded; only `stderr` continues to direct output back to the user. This is essentially a "fire and forget" process mode; it's quite handy if your program needs to spawn its own sub-demons, one for each incoming request. Note that an **`<async>`** minion needs to do all of its own file operations; therefore it can be convenient to create named pipes for each **`<async>`** minion.

SYSCALL Examples

Now it's time for a few examples. Here's a SYSCALL that runs ls on /var/spool and prints out the results. Notice that we must **`\`**-escape the '**`/`**' characters in **`/var/spool`** to **`\/var\/spool`**

```
        {
                window
                syscall /ls \/var\/spool / () (:_dw:)
                output  /:*:_dw:/
        }
```

The results are as we might hope:

```
# crm syscall_1.crm
anacron
at
clientmqueue
cron
cups
lpd
mail
mqueue
repackage
samba
squid
up2date
uucp
uucppublic
vbox
#
```

We can use SYSCALL to go off and grab a website, and scan it for anything interesting. Here, we'll use `lynx --dump` to grab the current www.slashdot.org website and check to see if it mentions anything matching the **`:_arg2:`** command-line argument:

```
{
        window
        syscall () (:_dw:) /lynx --dump www.slashdot.org /
        match /:*:_arg2:/
        output / Hey, there's news on :*:_arg2:\n/
}
```

Of course, this program's results will vary depending on what's on Slashdot today, but as of this writing, it yields:

```
# crm syscall_3.crm Voldemort
# crm syscall_3.crm Smith
# crm syscall_3.crm Neo
# crm syscall_3.crm SpaceShipTwo
 Hey, there's news on SpaceShipTwo
# crm syscall_3.crm Mars
# crm syscall_3.crm NASA
 Hey, there's news on NASA
```

```
        #
```

which we might suspect at least mildly useful to someone, somewhere..

Overriding Minion Process stdin and stdout

You can use the `bash` redirection operators in a SYSCALL. For example, to couple `stdout` directly to `stderr` (so it comes directly to your terminal):

```
        crm '-{ syscall /ls -la 1>&2/ }'
```

gives a normal ls -la listing (omitted here for brevity). Note that there must be no spaces between in the string **`1>&2`** for this to work (see the `bash` man page for details)

You can also redirect `stdin` and `stdout` into the file system, which includes not only normal files but also character-mode devices (in `/dev`) and named pipes (by convention these are usually in `/tmp`). This works both for `bash` commands and internal **`:label:`** calls, and for **<** (redirecting stdin) **>** (redirecting stdout) and **>>** (redirecting stdout with append) operations. Like the above redirection trick, you can't have any spaces between the redirection characters and the filenames.

Redirection of `stdin` and `stdout` does work for **`<async>`** minions; any pipe or file opened by redirection stays open unlike the default `stdin` and `stdout`, which are closed and discarded in **`<async>`** minions.

- Don't forget to \-escape any forward-slashes in filesystem names !
- On *NIX systems, you can usually open the TTY that "owns" the current process by specifying the "special" file **`/dev/tty`** device for the minion's input file, output file, or both. Don't forget to escape the forward-slashes as in using something like /\/dev\/tty/. Yes, we know it looks ugly.
- If you want to create semi-durable named pipes, use the `mkfifo` command; it's usually a good idea to make these pipes in the **`/tmp`** directory.
- You can use the **`:_pid:`** (current process PID) and **`:_ppid:`** (parent of current process's PID) variables to create lockfiles and unique pipenames as needed.

As a simple example of code that repeatedly spawns a process to read a pipe, try this:

```
        {
```

```
    window
    :loop:
    syscall /:my_local: <\/tmp\/my_pipe >>\/dev\/tty /
    goto /:loop:/
    exit
}
{
    :my_local:
    output /My pid: :*:_pid: and ppid: :*:_ppid: \n/
    input
    accept
    output /Done! \n/
    exit
}
```

This program repeatedly spawns a minion process to read the pipe /**tmp/my_pipe**, and output the result back onto the controlling TTY. Because we didn't specify **<async>** operation, the main program waits for the minion **:my_local:** routine to finish before it fires off another minion.

This example needs to be run with two terminals – one to actually run the program, and another to actually put text into the fifo we create with `mkfifo()`:

--Terminal 1--

```
# mkfifo /tmp/my_pipe
# crm fork_redirect.crm
My pid: 14325 and ppid: 14324
hello
this is a test
that's all
Done!
My pid: 14327 and ppid: 14324
putting more in
to the fifo
Done!
ctrl-C to exit
```

--Terminal 2--

```

# cat >/tmp/my_pipe
hello
this is a test
that's all
ctrl-D for EOF
# cat >/tmp/my_pipe
putting more in
to the fifo
ctrl-D for EOF
#
```

Note that the process ID of the spawned pipe-reading process continued to increase, but the PPID (Parent PID) remained constant.

If we had specified **<async>** then the main program would have fired off minions as fast as possible, which may or may not be what you want to do; it's handy to handle multiple network connections or massively parallel processors on a many-way SMP machine, but it's also a good way to stress out your kernel process tables.

True Recursive SYSCALLing

Because the caller and callee in a SYSCALL are entirely separate, we can write a recursive routine without worrying about name interference. In this example, we'll calculate factorial of whatever is in the data window. To demonstrate that we get copies of the data window and the isolated variable set, we'll pass the downward argument in the data window, do the math, and return the result upward in the output variable (yes, we're ignoring what happens if you put a non-integer into the data window – try it to see what happens if you get it wrong).

```
{
      output / Calculating :*:_dw: factorial by recursion \n/
      isolate (:out:)
      isolate (:status:)
      isolate (:local_x:)
      syscall <keep> /:my_factorial:/ () (:out:) (:status:)
      output /Returned output = :*:out:\n/
      output /Returned status:  ':*:status:'\n/
      exit
}
{
      :my_factorial:
      output [stderr] /In syscall; :_dw: is ':*:_dw:'\n/
      {
            #   recursion termination — is the argument < 1?
            eval  /:@:  :*:_dw:  < 1 :)/
            output [stderr] / Recursion bottomed out on a < 1 \n/
            output / 1 /
            exit
      }
      #      not time to terminate — arg > 1, so remember our
      #      arg and evaluate for arg = arg — 1.
      alter (:local_x:) /:*:_dw:/
      eval (:_dw:) /:@: :*:_dw: - 1 :/
      output [stderr] / and now do factorial of ':*:_dw:' \n/
      syscall <keep> /:my_factorial:/ () (:out:) (:status:)
      #    now :out: is factorial of our arg-1, we need to
      #    multiply by local_x and return.
      eval (:out:) /:@: :*:local_x: * :*:out: :/
      output [stderr] / My factorial is :*:out:, returning\n/
      #     and send out our result to our parent process
      output /:*:out:/
      exit
```

```
        }
```

When we run this code, we get:

```
        # crm syscall_2.crm
        5
        return, then ctrl-D
         Calculating 5
         factorial by recursion
        In syscall; :_dw: is '5
        '
         and now do factorial of '4'
        In syscall; :_dw: is '4'
         and now do factorial of '3'
        In syscall; :_dw: is '3'
         and now do factorial of '2'
        In syscall; :_dw: is '2'
         and now do factorial of '1'
        In syscall; :_dw: is '1'
         and now do factorial of '0'
        In syscall; :_dw: is '0'
         Recursion bottomed out on a < 1
         My factorial is 1, returning
         My factorial is 2, returning
         My factorial is 6, returning
         My factorial is 24, returning
         My factorial is 120, returning
        Returned output = 120
        Returned status:  'MINION PROC PID: 532 from-pipe: 6 to-pipe: 5'
        #
```

which is what we were hoping for: **`120`** as a result.

If you notice that in the first syscall, the '5' has an extra linefeed, that's because we put it there when we hit return, then ctrl-D. It isn't carried through because the linefeed is whitespace and whitespace goes away in a **`:@`** math evaluation.

KEEPing a long-running SYSCALL process

One issue with SYSCALLs is that there's no easy way for any particular program to tell if the program at the other end of a pipe is computing, waiting on a slow network site, or just plain hung[17]. The SYSCALL will simply wait until the process has piped out all it has to give

17 Not quite the classical halting problem- but just as bad.

(returning EOF), or the process output hits the anti-DoS limit (default is 2 megabytes).

Sometimes this wait is perfectly acceptable; sometimes it isn't. For example, consider a process that might be a literally endless fountain of data, but at a low rate (example: using `tail -f` on the system log, usually in `/var/log/messages`). It would be much better if we could get "what we can" quickly, go off and process the partial results, and then revisit the same process later to get any additional data that might have become available. The **`<keep>`** flag does exactly this – the minion program is "kept around" and later SYSCALLs with the same `:status:` variable will be able to continue communications with the minion.

Here's an example of looking at the system logs (right now, we're just going to print out anything that comes through – of course, once the text is in your CRM114 program, we can use any of our tools to scan it or manipulate it).

We don't have to do anything special beyond specifying **`<keep>`** and reusing the same status variable. Good programming practice says we shouldn't loop as fast as we can (which will result in a high CPU load without any real progress being made); hence the SYSCALL to `sleep()` after we output. CRM114 keeps track of all of the behind-the-scenes process management issues. Again, we have to **`\`**-escape the forward slashes in the filename string `/var/log/messages`. Here's working code to demonstrate how to **`<keep>`** your way on a slow but infinite data source (like a log file):

```
{
      window
      isolate (:my_tailings: :my_status:)
      syscall <keep> () (:my_tailings:) (:my_status:) \
            / tail -f \/var\/log\/messages /
      output /Log entry of:  ---- \n ':*:my_tailings:' \n ----end----\n/
      syscall /sleep 1/
      liaf
}
```

When we run it (as a user with privilege to read `/var/log/messages`) , we get (over a span of several minutes):

```
      # crm syscall_4.crm
      Log entry of:  ----
       'Jan  3 12:00:02 katsuragi crond(pam_unix)[962]: session closed for
      user root
      Jan  3 12:00:05 katsuragi crond(pam_unix)[963]: session closed for
      user root
      Jan  3 12:01:01 katsuragi crond(pam_unix)[992]: session opened for
      user root by (uid=0)
      Jan  3 12:01:02 katsuragi crond(pam_unix)[992]: session closed for
```

```
user root
Jan  3 12:05:01 katsuragi crond(pam_unix)[995]: session opened for
user root by (uid=0)
Jan  3 12:05:05 katsuragi crond(pam_unix)[995]: session closed for
user root
Jan  3 12:10:01 katsuragi crond(pam_unix)[1002]: session opened for
user root by (uid=0)
Jan  3 12:10:01 katsuragi crond(pam_unix)[1003]: session opened for
user root by (uid=0)
Jan  3 12:10:02 katsuragi crond(pam_unix)[1002]: session closed for
user root
Jan  3 12:10:06 katsuragi crond(pam_unix)[1003]: session closed for
user root
'
 ----end----
Log entry of:  ----
 'Jan  3 12:13:51 katsuragi cardmgr[1790]: + ./ide: line 34:
/sbin/ide_info: No such file or directory
'
 ----end----
Log entry of:  ----
 'Jan  3 12:13:52 katsuragi fstab-sync[1121]: removed mount point
/media/idedisk1 for /dev/hde1
'
 ----end----
```

which we can now use for anything we please. (take note of this example if you want to write a system monitoring program.)

ASYNChronous “fire and forget” processes

In some cases, we don't want to hear from the minion process again; we simply want to launch the minion and then let go of it. The **`<async>`** flag does exactly this – the minion is launched, we feed it any specified `stdin`, grab any immediate output, and the subprocess now can run to completion on its own.

Good places to use **`<async>`** are for processes that are spawned on an as-needed basis. This can be a big speedup for things like mail servers; instead of re-invoking process loading, JIT compilation, etc. a “hot copy” of the process can be forked for each incoming request. With careful design, many minions can be invoked per second, with essentially zero JIT compilation time needed..

Here's an example – this program will fork 10 minions 100 times (that's 1000 minions); each minion will simply exit. On the base machine this test was run on (a 933 Mhz

TransMeta laptop with 512MBytes of RAM), the fastest CRM114 program possible takes about 66 milliseconds of clock time, and about 10 milliseconds each of user time and system time:

```
# time crm '-{ window; exit}'

real    0m0.066s
user    0m0.009s
sys     0m0.008s
#
```

Here's our benchmark beast-with-a-thousand-young code:

```
{
      window
      isolate (:count:) /100/
      {
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            syscall <async> /:just_exit:/
            eval (:count:) /:@: :*:count: - 1 :/
            eval / :@: :*:count: > 1 :/
            liaf
      }
}
:just_exit:
exit
```

This program doesn't do anything useful (except perhaps stress-test the kernel's ability to handle floods of processes. For reasons explained below, this process will actually fork 3000 processes (1000 true minions, plus 2000 assist processes.). Let's time it:

```
# time crm asynctest_1.crm

real    0m2.245s
user    0m0.355s
sys     0m1.733s
#
```

That's 0.35 milliseconds of user time and 1.7 milliseconds of system time per minion. This is very high performance, considering that the test machine is a sub-GHz subcompact Transmeta laptop running on batteries.

To test that we actually did run all of those processes, let's do a trivial amount of output in each of the SYSCALLs.

```
      {
            window
            isolate (:count:) /100/
            {
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  syscall <async> /:just_exit:/
                  eval (:count:) /:@: :*:count: - 1 :/
                  eval / :@: :*:count: > 1 :/
                  liaf
            }
      }
      :just_exit:
      output <append> [ async_output.txt ] /X/
      exit
```

This will put a single X at the end of async_output.txt for each minion (all 1000 of them), plus one for the main program when it finally exits.

When we time this monstrosity we get:

```
      # time crm asynctest_2.crm

      real    0m2.735s
      user    0m0.449s
      sys     0m2.118s
      #
```

Not bad either for 1000 subprocesses – it took 0.45 milliseconds of user time and 2.2 milliseconds of system time per process to fork, open a file, `lseek()` the file to the end, write to the file, close the file, and exit.

How many X's did we actually do?

```
        # wc async_output.txt
          0    1 985 async_output.txt
        #
```

We only have 985 X's in the output file! (if you do this example, your number may differ) There should have been 1001 X's. What happened?

The answer is that *NIX file I/O is not exclusive; just because one process has a file open does not mean that other processes are excluded from also writing that same file. Indeed, even “mandatory” file locks (such as created by `fcntl()` , in contrast to the truly advisory locks that `flock()` creates) are only advisory in nature unless the disk volume was mounted with mandatory locking enabled (such as **`mount -O mand`** , which is not the default configuration for most *NIX distributions).

Moral to remember – if you will be spawning multiple programs, each of which will be doing file I/O to the same files, you must either make sure that the disk volumes are mounted with mandatory locking enabled, or your program must use a subterfuge such as application-based lock files to assure sequential access.

Remember that **`:_pid:`** can be your best friend in such situations. Also remember that even if you write a good locking-file implementation, NFS network mounts will often cache file and directory I/O for an unbounded length of time, which may defeat even perfect file-locking semantics.

Let's see how fast a bash program can run compared to native CRM114. We'll test this by running a shell **`exit`** command instead of doing file I/O. Here's the source code:

```
        {
                window
                isolate (:count:) /100/
                {
                        syscall <async> /exit/
                        syscall <async> /exit/
                        syscall <async> /exit/
                        syscall <async> /exit/
                        syscall <async> /exit/
                        syscall <async> /exit/
                        syscall <async> /exit/
                        syscall <async> /exit/
```

```
                  syscall <async> /exit/
                  syscall <async> /exit/
                  eval (:count:) /:@: :*:count: - 1 :/
                  eval / :@: :*:count: > 1 :/
                  liaf
            }
      }
      exit
```

The resulting timings are:

```
# time crm asynctest_3.crm

      real    0m11.731s
      user    0m4.105s
      sys     0m7.146s
      #
```

showing that bash is considerably slower than CRM114's JIT-accelerated code. Here's a summary table of our timings (per spawned program, in milliseconds, on a sub-Ghz Transmeta laptop running on batteries):

Spawned Program	*System CPU Time*	*User CPU Time*	*Clock Time*
`having bash` **`exit`** (CRM114 not used)	8.000 msec	9.000 msec	66.000 msec
using SYSCALL `bash` **`exit`**	7.146 msec	4.105 msec	11.731 msec
SYSCALL "open, write, close, exit"	2.118 msec	0.449 msec	2.735 msec
SYSCALL "CRM114 exit"	1.733 msec	0.355 msec	2.245 msec

From this we can see that it is much faster to use SYSCALL to create a minion CRM114 process versus creating a process from the `bash` command line. If we can stay inside CRM114, the overhead drops by roughly an order of magnitude. This is because the FORKed process doesn't have to re-load the program, re-execute the preprocessor and rerun the JIT startup phase.

The actual action behind SYSCALL is a little more complicated than a simple fork. We fork both the actual minion, plus a “pusher process”, to do the actual pumping of input into the minion's stdin. This is how we avoid a deadly embrace where both master and minion have written something and are now waiting for the other process to do a read and make room in the buffer so that both processes can write more.

With an <async> process, it gets even more complex- we have to fork the minion and the pusher process, but after we've read any immediate output, we need to fork a “sucker” process to drop any further minion stdout in the bit bucket, otherwise the minion itself will block waiting for its output buffer to be emptied.

The net result is that every SYSCALL launches at least two processes (minion and pusher), and <async> SYSCALLs may launch three (minion, pusher, and sucker). Keep this in mind when you think about having hundreds of simultaneous minion processes; you may actually run out of process slots doing things that don't seem to be kernel-resource edgeplay at first glance.

Inserting blocks of code with INSERT

Good software engineering suggests that localizing the common configuration items into a single file is a good thing. Putting commonly used code into reusable files is also a good thing (hopefully the commonly used code is well-debugged). The typical way these configuration and common-reuse files are used is with a INSERT statement.

INSERT is somewhat different than all of the other CRM114 statements – it is a *preprocessor* statement that causes changes in the pre-JIT source code.

For example:

```
{
      insert mathdemo2.crm
}
```

will insert the contents of the file `mathdemo2.crm`, yielding the intermediate code below (the command line flag to generate this level of detail in a listing is **`-l 3`** (that's a lowercase L).

In this level 3 listing, the first number on each line is the line number; the second number in **`{}`** is the nesting level. The actual statement follows the colon. Listing levels 4 and 5 show even more detail including FAIL, LIAF, and TRAP targets, and the JIT parse information:

```
# crm -l 3 insert_1.crm

0000 {00} :
0001 {00} :
0002 {00} :  {
0003 {01} :    insert=mathdemo2.crm
0004 {01} :    {
0005 {02} :      eval (:_dw:) / :@: :*:_dw: : /
0006 {02} :      output /:*:_dw: \n/
0007 {01} :    }
0008 {01} :
0009 {00} :  }
0010 {00} :
0011 {00} :
0012 {00} :
```

which executes just as `mathdemo2.crm` itself did:

```
# crm insert_1.crm
2 + 3 * ( 6 * 7)
 210
#
```

There are several things to note here- when the preprocessor expanded the INSERT, it changes the line reading:

```
insert mathdemo2.crm
```

to read:

```
insert=mathdemo2.crm
```

(yes, the preprocessor added an equals sign). This shows in the listing that the INSERT file was found. We'll talk about what happens if the file wasn't inserted successfully later in this chapter,

INSERTing inside an INSERT file

INSERT files themselves can perform INSERTions; the actual evaluation is recursive. No special syntax is needed; the sub-files are inserted in the order specified.

For example, here's three files; `inserttest_a.crm` inserts `inserttest_b.crm`, and `inserttest_b.crm` inserts `inserttest_c.crm`.

```
Here are the files;
# cat inserttest_a.crm
output /\nStart of insert processor testing \n/
output / start here ... \n/
insert inserttest_b.crm
output / and the last bit of normal text \n/

# cat inserttest_b.crm
output /    the first middle bit... \n/
insert inserttest_c.crm
output /    the last middle bit... \n/

# cat inserttest_c.crm
output /       the really middle-middle bit ... \n/
```

```
#
```

When we generate a listing with **-l 3**, we get:

```
# crm -l 3 inserttest_a.crm

0000 {00} :
0001 {00} :  output /\nStart of insert processor testing \n/
0002 {00} :  output / start here ... \n/
0003 {00} :  insert=inserttest_b.crm
0004 {00} :  output /    the first middle bit... \n/
0005 {00} :  insert=inserttest_c.crm
0006 {00} :  output /       the really middle-middle bit ... \n/
0007 {00} :
0008 {00} :  output /    the last middle bit... \n/
0009 {00} :
0010 {00} :  output / and the last bit of normal text \n/
0011 {00} :
0012 {00} :
```

showing that the INSERT files are inserted verbatim (there is no need to add extra {} brackets to "set off" the INSERTed text. In fact, INSERTs don't even need balanced **{}** brackets; it's perfectly reasonable to write a pair of INSERT files to add a special processing preamble and postamble to a block of code (say, to wrap arbitrary user code inside a TRAP).

When we run this code, we get:

```
# crm inserttest_a.crm
return, then ctrl-D
Start of insert processor testing
 start here ...
    the first middle bit...
       the really middle-middle bit ...
    the last middle bit...
 and the last bit of normal text
```

which shows that we did indeed descend through all three INSERT files in the proper nested order.

Missing INSERT files

What if the INSERT file was missing? Let's try it:

```
      {
            insert There_is_no_such_file.crm
      }
```

which yields (after preprocessing):

```
      0000 {00} :
      0001 {00} :  {
      0002 {01} :    insert=There_is_no_such_file.crm
      0003 {01} :
      0004 {01} :    ###### THE NEXT LINE WAS AUTO_INSERTED BECAUSE THE
      FILE COULDN'T BE FOUND
      0005 {01} :    fault /Couldn't insert the file named
      'There_is_no_such_file.crm' that you asked for.  This is probably a
      bad thing./
      0006 {00} :  }
      0007 {00} :
      0008 {00} :
```

There's some funny business around line 4 and and 5; the preprocessor couldn't find our (intentionally) missing file, so it inserted a FAULT statement instead. Although you haven't seen a FAULT statement, you can probably guess that it forces a runtime error at that point (and you'd be right – it does – and the error is the FAULT command's **`/slashed string operand/`**.

When we actually run the program, we get:

```
      # crm insert2.crm
      ( 2 * 4) + 5

      crm: *ERROR*
       Your program has no TRAP for the user defined fault: Couldn't
      insert the file named 'There_is_no_such_file.crm' that you asked
      for.  This is probably a bad thing.
       Sorry, but this program is very sick and probably should be killed
      off.
      This happened at line 5 of file insert2.crm
      The line was:
      --> fault /Couldn't insert the file named
      'There_is_no_such_file.crm' that you asked for.  This is probably a
      bad thing./

      #
```

As expected, we get an error message, telling us that we couldn't insert the file we asked for. But there's another implication- the error message starts out "`Your program has no TRAP for....`" . This implies that we can somehow trap errors, even preprocessor time errors.

Indeed, CRM114 has a very powerful lexically-scoped fault trapping mechanism. We will cover a full explanation of FAULT and TRAP later. For now, accept that many preprocessor error messages you get will start out with "Your program has no TRAP for..." and relax knowing that you will soon know how to deal with these errors at runtime.

UNION and INTERSECT

UNION and INTERSECTion are operations on non-ISOLATEd vars; they set a new variable to the UNION or INTERSECTion of two or more strings.

The syntax is unsurprising:

```
UNION (:out_var:) [ :in_var_1: :in_var_2: ...]
INTERSECT (:out_var:) [ :in_var_1: :in_var_2: ...]
```

The **`in_var`** variables must all be parts of the same top level variable (either the **`:_dw:`** data window or the same ISOLATEd variable. In some versions of CRM114, only the data window is supported.

The definition of INTERSECTion is unsurprising:

> INTERSECTion – return the longest string segment that is a member of each input var.

This is handy because it lets you use several regexes to grab different “candidate” strings, then INTERSECT them to see if anything meets all the requirements of each regex. The resulting output var is non-ISOLATEd- it's as though you handcrafted a regex that combined each of the candidate regexes with order independence.

The definition of UNION is a little different:

> UNION – return the shortest string that is either in an in-var, or between any two characters of any of the input vars.

Note that although the result of a UNION is always non-ISOLATEd, the “between any two” clause means it may contain text that was not a member of any of its in-vars. Rather than a liability, this is quite powerful.

Consider a situation where a set of N markers may exist in a file; the text between these markers is the text of interest. A pure regex based solution to this requires at O(n^2) operations just for pairwise consideration of the markers, O(n^3) for consideration of all triplet markers, and so on.

Here's a (somewhat contrived) example of INTERSECTion and UNION.

```
{
    window
    alter (:_dw:) / a b c d e f g h i j k l m n o p q r s t u v w x y z /
    output /We start with: ':*:_dw:' :*:_nl:/
    match (:alpha:) /a/
    match (:lima:) /l/
    match (:sierra:) /s/
    match (:z:) /z/
    match (:abc:) /a b c/
    match (:cde:) /c d e/
    intersect (:t1:) [:abc: :cde:]
    output /intersection of abc and cde is t1: ':*:t1:' :*:_nl:/
    union (:t2:) [:lima: :sierra:]
    output /union of l thru s is t2: ':*:t2:' :*:_nl:/
    intersect (:t3:) [:abc: :t2:]
    output /intersection of abc and t2 is t3: ':*:t3:' :*:_nl:/
    union (:t4:) [:z: :t1:]
    output /union of z and t1 is t4: ':*:t4:' :*:_nl:/
}
```

Here's the result:

```
We start with: ' a b c d e f g h i j k l m n o p q r s t u v w x y z '
intersection of abc and cde is t1: 'c'
union of l thru s is t2: 'l m n o p q r s'
intersection of abc and t2 is t3: ''
union of z and t1 is t4: 'c d e f g h i j k l m n o p q r s t u v w x y
z'
```

UNION and INTERSECT are only occasionally used in CRM114, but when they are used, they're lifesavers.

LEARN and CLASSIFY

One of the biggest reasons people might choose CRM114 for a particular project is that CRM114 integrates several very powerful classifiers oriented to classify natural languages, texts, and byte streams, and is designed to make it easy to add more classifiers in the future.

The syntax of LEARN and CLASSIFY is:

```
learn <flags> [in_text] (stats_file) /tokenize_regex/

classify <flags> [in_text] (stats_file | stats_fileN) (:out_var:)
/tokenize_regex/
```

Because it's easy to add classifiers to CRM114, this is the one chapter of the book most likely to be out of date. If something isn't working for you (or not working well enough) **your first stop should be the documentation that came with your particular version of CRM114.**

At this writing, CRM114 has seven different classifiers, some with multiple variants, and that number is sure to increase in the future. Some of these classifiers are two-way (they can make only a yes/no choice); others can do an N-way choice (up to some compile time limit, typically 128 different classes). N-way choices can be useful as they can split your mail into multiple categories like “spam”, “party invites”, “conference correspondence”, “work memos”, and “hot dates”.

We'll also need to touch on the difference between the training method (that is, when and how a particular piece of new information should be added to the database), and learning methods (how the database itself is modified on learning and utilized on classification).

LEARN and CLASSIFY flags.

The **`<flags>`** options are in two categories: to select which classifier to use, and to set various options in those classifiers. The classifier selection flags are:

`none` - use Markovian classification

`<osb>` - use OSB classification

`<osb unigram>` - use plain Bayesian classification

`<osbf>` - use OSBF classification

`<winnow>` - use Winnow classification

`<correlate>` - use Correlator classification

`<hyperspace>` - use hyperspatial classification

and use these classifier modifiers:

`<unique>` - learn or classify only with unique features – second and further repeats of a feature in the document are disregarded as far as the current statement is concerned. Thus, repeated words and phrases become "invisible" to the classifier.

`<refute>` - "undo" this learning. This is used both in undoing mistakes, and for training "not this class" in double-sided training. A few classifiers don't allow this kind of undoing.

`<microgroom>` - use an automatic statistics-file management system to "age out" old data from the statistics files

`<unigram>` - use unigrams only, rather than the much-more-extensive OSB or Markovian feature systems. This is only valid for the OSB, Winnow, and Hyperspace classifiers; the other classifiers will silently disregard it.

We'll describe these classifiers below, with their strengths and weaknesses.
There are many reasons for this plethora of classifiers in CRM114 – first, it's been mathematically proven[18] that generalized machine learning is "not well-ordered"; there is no one optimizer that always beats every other optimizer on every possible test set of a domain; an embarrassing conclusion to some theoreticians, but it's an engineering reality.

18 Wolpert and Macready, the "No Free Lunch" theorem, IEEE Transactions on Evolutionary Computation, April 1997, pgs 67-82, showing that no single algorithm beats every other algorithm over all sources.

A second reason for multiple classifiers "on tap" is that many of the applications of CRM114 are "moving targets"- spam filtering in particular. As spammers improve their techniques to circumvent filters, the classifiers themselves are improved by their authors.

A third reason for multiple classifiers is the meta-result of reason 1- not only is there no one best classifier, there is also not one best classifier *author*. By making it possible to integrate new classifiers into the CRM114 base system, upgrading classifiers no longer means constructing an entire new system. With CRM114 most of the infrastructure can remain untouched when the classifier is upgraded; this makes it easy to test and upgrade classifiers while not re-coding the entire application.

Classification Output

When the CLASSIFY statement executes, it returns its result in two ways:

- the **`status result`** variables
- a FAIL skip (or not)

Both of these outputs are optional.

When the CLASSIFY statement runs, each of the N statistics files is compared with the incoming text and scored. The highest-ranked statistics file is deemed to be the "best match", and the optional **`:stats:`** variable is modified to contain the actual results (the variable doesn't need to be named **`:stats:`**, but that's a very good convention to follow so that other programmers will be able to understand your code.).

In addition, the cumulative statistics of all of the statistics files to the left of the divider **"|"** are compared to the cumulative match statistics of all of the statistics files to the right of the divider. If the left side files (as a group) are a better match than the right side files (as a group) then the CLASSIFY statement "succeeds", and execution of your program continues. If the right side files are a better match (or the matches are of equal quality), the CLASSIFY statement does a FAIL to the end of the current **`{}`** block. We FAIL on a no-conclusion result because of the CRM114 motto- "When in doubt, nothing out!"

As we've mentioned, both of these output methods are optional. If you don't want to use a status results variable, don't put one in.   If you don't want the CLASSIFY statement to FAIL, don't put any statistics filenames to the right of the divider vertical bar (you can omit the divider bar if you want); that will assure that the left-side files will always be a better match than the (nonexistent) right side files.

Training Methods

All of the current CRM114 classifiers are *learning* classifiers; they are fed a set of known examples (called a *corpus*), and some algorithm is used to produce stored intermediate statistical information (typically stored on disk). This intermediate state is used by another part of the algorithm to classify incoming unknown texts. These new texts are classified and spot-checked; any mistakes are fed back into the learning algorithm and become part of the corpus. What particular set of texts is fed back into the learning algorithm is controlled by the *training method*.

Training methods are not something that's "part of the classifier"- rather, it's how you train the classifier to become more accurate. Therefore, there are no **<keywords>** for training; rather you implement your training method as part of your overall system design. If you can, we recommend letting individual users train their own classifiers; this yields the highest accuracy (but means that the user interface needs some way for mere mortals to feed information back into the classifier). Some installations have found this unnecessary; they allow only a small core of employees to define the training corpus and then that corpus is used for millions of users.

TET – Train Every Thing

The most obvious training method is to keep statistics on every known text, without any sort of optimization. This is called Train Every Thing (TET) and it actually turns out to be a fairly bad training method. In numerous benchmarking tests, TET consistently gave the worst results, no matter what specific classifier was used. This interesting result is not completely explained, but it is clear that TET not only stores a lot of irrelevant data (using up valuable space in the statistics files), but that a lot of the relevant data is redundant. So, put the idea of training with TET out of your mind; TET is almost always the wrong answer, no matter what the question. [19]

TOE- Train Only Errors

A much better training method is to Train Only Errors (TOE). In TOE learning, statistics are accumulated only when the classifier makes an error. This means that before any example text is actually trained, the classifier is run to see if the text would have been correctly classified with the current statistics information. Only if the classifier makes an error is the text trained.

19 There are some machine learning techniques, such as Support Vector Machines (SVMs) that can be given a full (TET-style) corpus, and will auto-select only the examples that are pertinent to the class separation problem. However, these techniques can be extremely CPU-intensive during the auto-selection phase.

Usually, the statistical features of the mis-classified text are accumulated into the class that the message should have been classified into (but see DSTTT below). The other classes' accumulations aren't affected. TOE training is consistently “pretty good”; various other training methods can work better on one or two of the specific classifiers (and we'll note those below!) but TOE always works “pretty well”. Remember – TOE means Train Only Errors; no matter how narrow the margin, if a text was classified correctly, it's not trained.

SSTTT – Single Sided Thick Threshold Training

For some classifiers (but not all!) TOE can be improved by Single-Sided Thick Threshold Training (SSTTT). In SSTTT, we train if there's an error the same way as in TOE, but even if the correct class was chosen we check to see how strongly correct the right class was. If the margin of correctness isn't high enough, the text is trained into the correct statistics file anyway. Just as in TOE, the only statistics that are changed are those of the correct class; it's just that we train not just errors, but also train classifications if they not sufficiently correct.

DSTTT – Double Sided Thick Threshold Training

The next training method is Double Sided Thick Threshold Training (DSTTT). In DSTTT, we train into the correct class if the classifier wasn't sufficiently sure of itself, and we train out of every incorrect class that the classifier wasn't sufficiently sure was the wrong class. Double-Sided Thick Threshold is the usually the best training method for the Winnow classifier, but usually not for any of the other classifiers.

DSTTTR – Double Sided Thick Threshold Test and Reinforce Training

DSTTTR (also sometimes known as TOER) is a training method that can be described as “smart DSTTT”. We start with the same procedure as SSTTT – if an incoming unknown text isn’t scored correctly or scored correctly but not correctly enough (typically, by a margin of at least 10 to 20 pR units) it gets trained just as in SSTTT. Then the text is re-classified again, and the improvement measured. If the new classification doesn’t meet the threshold requirement, or didn’t improve in the correct direction by at least some smaller margin (typically 3 pR units) then the text gets trained out of the incorrect classes. The difference here is that in straight DSTTT, the train-out-of action is based solely on the before-training pR values, but in DSTTTR the decision to proceed with training out-of class is determined by the pR value after training and retesting. The reports are that DSTTTR works even better than TOE in most circumstances.

TTE - Train Till Exhaustion

Train Till Exhaustion – TTE – is a meta-training method; it means after we train a new text into the classifier, we immediately re-classify the new text to see if it was correctly classified (including checking margin of correctness for SSTTT and DSTTT). If the text isn't “adequately learned”, we train it again. This process repeats until we either run out of time or patience. We don't move on to another example text until we have this sample trained correctly. This is somewhat like DSTTTR but we only train “in”, not train “out”.

Because some classifiers cannot be guaranteed to be able to learn every possible text, it is wise to put a loop-count maximum on train-to-exhaustion training programs. Otherwise, your program may loop infinitely as it retrains a text but never quite succeeds in adequately learning the text.

Some variants of TTE keep the entire back list of any example that was ever trained into the system available, and the TTE loop goes through each and every one of those examples, training as necessary. This variation is particularly prone to fail to converge, so it's doubly important that you set a loop count limit if you use this method of TTE

TUNE – Train Until No Errors

Another “meta” training method is Train Until No Errors, or TUNE. A TUNE training means to repeat testing and training on all of the example texts until no more errors are made (or until some limit of time or patience has been exceeded). Since this means running the entire training corpus multiple times, it's usually not done as part of a real-time training system. (note the difference between TUNE and TTE with stored training corpus – in TUNE the entire known corpus is repeatedly tested and trained, while in TTE at most only the example texts that were classified incorrectly at least once stay in the loop for retraining)

For example, you can TUNE with TOE, or TUNE with SSTTT; generally speaking the accuracy of a TUNEd classifier is better than the accuracy of the same classifier, with the same base training method, but not TUNEd.

Although it's unlikely, it's possible to have a training set that never successfully TUNEs completely; any TUNEing code you write should have a limiter so that it doesn't run forever. The mathematical theory that determines whether a particular classifier can be TUNEd to perfect accuracy is beyond what we can cover here, but if you are really interested, you should start by looking up “linear separability” and work from there.

Current Classifiers in CRM114

At this writing, there are seven classifiers supported in CRM114. Markovian, OSB, OSB Unigram (which is exactly equivalent to a Bayesian classifier), OSBF, Winnow, Correlate, and Hyperspace. The default classifier is Markovian, and if a program doesn"t specify which classifier to use, it gets Markovian. We'll describe the basic principles of each of these algorithms, as well as the appropriate learning methods.

What's In A Word?

All of the current CRM114 classifiers except Correlate work with the concept of *words* . A word is the smallest chunk of text that would have meaning in the context of the natural language (that is, usually English, not the CRM114 "language"). For example, this sentence has seven words. Those words are "for", "example", "this", "sentence", "has", "seven", and "words".

To break an incoming text up into words, the LEARN and CLASSIFY statements use a regex. Each successive non-overlapping match of a regex is one token word for the classifiers. Text that doesn't match the word token regex is disregarded by the classifier. If you don't supply a word-tokenizing regex, CRM114 uses a default of **`[[:graph:]]+`** which defines a word as "something with ink, surrounded on both sides by whitespace". This default is generally pretty good; change it if you want to experiment.

Note that there's no constraint that a word be something that a human would call a "word"; it's part of CRM114's strength that a "word" is defined totally by the word regex, not by a preconceived human notion of "word". For example, an IP address in dotted-quad notation *might* be a word in some applications – or perhaps it might be four words (each of the octets separately) or perhaps it shouldn't be considered a feature at all (in which the word regex should be written to simply skip over the uninteresting parts of the text.)

Part of the differences between the classifiers in CRM114 is how each classifier turns a word of text into a *feature* – that is, something that has an actual entry in the statistics files. Unlike most of the simple Bayesian filters available, most CRM114 classifiers don't use single words as countable features; words are combined and only word combinations are significant.

Markovian Classification

The default classifier in CRM114 is *Markovian*. Markovian classification is an extension of the common Bayesian text classifiers that has extra code to bridge multiple words and phrases. A Markovian classifier attempts to match word phrases in the known texts with word phrases in the incoming unknown text. The current default Markovian classifier in CRM114 has a phrase window 5 words long, and looks not only at the individual words, it looks at word pairs, word triples, word quadruples, and word quintuples. It even looks at tuples with each word selectively ignored, each pair selectively ignored, and so on (placeholders are inserted where a word is ignored). In the end, within a five-word window, every possible combination of words, both used and ignored, is computed as an independent feature and checked against the statistics file for quality of match. This makes it a very powerful phrase matcher.

For example, the sentence “Houston, we've got a problem” would pass through the window of length 5 yielding the following sixteen phrases:

```
Houston
Houston we've
Houston <skip> got
Houston we've got
Houston <skip> <skip> a
Houston we've <skip> a
Houston <skip> got a
Houston we've got a
Houston <skip> <skip> <skip> problem
Houston we've <skip> <skip> problem
Houston <skip> got <skip> problem
Houston we've got <skip> problem
Houston <skip> <skip> a problem
Houston we've <skip> a problem
Houston <skip> got a problem
Houston we've got a problem
```

Notice that even significant variations of the initial phrase won't change many of the result phrases; that's one of the powers of CRM114's Markovian classification system. If you think of each of the phrases generated by this phrasing system, you can see that the form an irregular lattice, somewhat like a finite state machine. Each new word brings another set of possible transitions; how often we see that phrase defines how likely that transition is (math majors will recognize this as a Markov Random Field, with clique size = 5).

All of these resulting phrases are looked up in the statistics files, which give the probability that the phrase occurs in each of the classes. Given the individual phrase probabilities, the classical Bayesian probability chain rule is used to form a final probability value. The Bayesian chain rule itself is usually expressed as:

```
                                          P(seeing feature F,  given class C) *  P (in class C)
P (in class C, given feature F) = -------------------------------------------------------------------
```

Sum all classes **[** P(feature F, given class C) * P (in class C)**]**

Because the Bayesian chain rule is predicated on the inputs being statistically independent, and in the case of text analysis with overlapping windows the inputs are clearly not independent but highly correlated, the output probability is highly overemphasized – often being "sure" to several tens of 9's of probability. To bring this back into a human-understandable range, CRM114 introduces the concept of *pR* . We'll discuss pR in depth below- for now consider it an indicator of confidence. High positive pR means that a text is very likely in a class; negative pR means that the text is likely not in the class, and a pR within ten or so counts of 0 means "unsure".

Markovian classification works reasonably well with any of the simpler learning methods (TOE, SSTTT, TUNE), however it doesn't perform very well with DSTTT.

There's no particular keyword to activate Markovian classification; it's the default if you don't specify another classifier. Markovian classification is compatible with the **`<unique>`** flag which may or may not improve accuracy. You can also use the **`<microgroom>`** flag to automatically manage the overall use of space in the statistics files, and the **`<refute>`** flag to "un-learn" mistaken training cases (this undoes the damage done by a mistaken LEARN). For spam filtering applications, OSB or OSBF are usually (but not always) more accurate and faster than Markovian classification.

OSB Classification

OSB (Orthogonal Sparse Bigram) classification was built on Markovian; it was a variation to increase speed but it also turned out to usually (but not always) be more accurate and use less memory and disk space as well. The biggest simplification of OSB versus Markovian is that OSB uses only two words at a time in any given five-word window, instead of the one-through-five words that Markovian uses. The particular word pairings for each OSB feature form an orthogonal sparse basis set out of the bigrams, hence the term OSB.

In our previous example sentence "Houston, we've got a problem", Markovian came up with sixteen different phrases. OSB instead comes up with only four phrases – the four (not five) word pairs containing the first and second words, the first and third words, the first and fourth words, and the fourth and fifth words. Here are the four word pairs that OSB would generate:

```
    Houston we've
    Houston <skip 1 word> got
    Houston <skip 2 words> a
```

```
      Houston <skip 3 words> problem
```

After we look up these word pairs, we use the same Bayesian chain rule formula to combine the probabilities into an overall probability that the text belongs to one class or another.

Experimentally, OSB needs only about one-fourth of the statistics storage space needed by a pure Markovian classifier to achieve the same accuracy. (A careful reader might take the above to mean that OSB doesn't use single words – and that careful reader would be correct. OSB classifiers do <u>not</u> use single words, taken one at a time, as features. Extensive testing has shown that accuracy often <u>decreases</u> if single words are used in an OSB-style system! Who knew?)

OSB classification works acceptably with TOE, but it really improves with SSTTT and TUNEing; training all errors, plus training any text with a pR between +10 and -10 seems to be near optimal. Like Markovian, it doesn't work very well with DSTTT. OSB classification is activated with the **`<osb>`** flag, and is particularly accurate if used with the **`<unique>`** flag. You can also use the **`<microgroom>`** flag to automatically manage the overall use of space in the statistics files, and the **`<refute>`** flag to "un-train" mistaken training cases (this undoes the damage).

OSB Unigrams

One of the options that is only available on OSB, Winnow, and Hyperspace is the unigram feature set. This is turned on by using **`<osb unigram>`** as the control flag set.

Unigram means "single word", and this flag turns off all of CRM114's fancy multi-word bridging software. The result is a classifier that is a basic Bayesian system.

Note that we don't recommend this for most users. The reason for this option is to allow users to test the multiword classification methods and verify that they're getting an improvement over simple Bayesian single-word methods, without having to go build an entirely new system. As with plain OSB, you can use **`<microgroom>`**, **`<unique>`**, and **`<refute>`** with OSB Unigram.

OSBF Classification

OSBF is a derivative of OSB classification, with "double extra voodoo", (that is, the text features are the same as the orthogonal sparse bigrams of OSB, but some of the mathematical evaluation constants are determined in an ad-hoc way). Additionally, OSBF uses a relative of TF-IDF (Term Frequency – Inverse Document Frequency) called EDDC (Exponential Differential Document Count) system to determine when a particular feature is “worthy” of use in a classification. At this point, OSBF should be considered “production ready”, as is OSB. Many classifier improvements are first tested and tuned in OSBF and then back-ported to the other classifiers if they actually improve accuracy or speed. Some important improvements that made the jump back into the other “production” classifiers are positional microgrooming and document-based confidence calculation.

OSBF is (at this writing) one of the fastest and most accurate “classical” classifiers in the CRM114 set (the lead goes back and forth between OSBF and Hyperspace). That said, the voodoo inherent in EDDC can cause problems; OSBF has been known to fail to converge on a learning set. Use OSBF only if you are willing to accept the risks inherent in the code.

OSBF is closely related to OSB, and it trains similarly- TOE with SSTTT works very well, and TUNEing works even better, and DSTTTR (a.k.a. TOER) best of all. Like OSB, it works best when trained on any error, and on any text that gives a pR between -10 and +10.

The OSBF classifier is activated with the <**`osbf>`** flag, and always uses **`<unique>`** mode, even if you don't specify the **`<unique>`** flag. You can also use the **`<microgroom>`** flag to automatically manage the overall use of space in the statistics files and the **`<refute>`** flag to “un-train” mistaken training cases (this undoes the damage done by an incorrect TRAIN).

Winnow Classification

Winnow is based on Lightstone's WINNOW algorithm, and as such, is not truly a probabilistic algorithm at all. Instead of generating probabilities based on seen-feature counts, Winnow initially weights all features at a uniform value of 1.0000 and then applies promotion and demotion factors whenever a feature is found in a correctly or incorrectly categorized document. At this writing, promotion is equivalent to multiplying the feature's weight by 1.23, and demotion is equivalent to multiplying by 0.83 . Like OSB and OSBF, Winnow uses the orthogonal sparse bigram word-pairs as input features. To evaluate a document's overall class, the feature weights are summed (not multiplied !); the class with the highest total score is the nominal “best class”. This is a simple perceptron with back-propagation; as such, Winnow cannot separate anything that isn’t linearly separable (although the OSB feature set on the front end itself produces inputs that can do linear separability as long as the features of interest are no more than five words apart.

In terms of accuracy, Winnow is usually comparable to OSB. Winnow works <u>only</u> with DSTTT and <u>**must**</u> be trained that way; typically the unsure domain is -20 pR to +20 pR for best behavior.

The Winnow classifier is activated with the **`<winnow>`** flag, and always uses **`<unique>`** mode, even if you don't specify the **`<unique>`** flag. You can also use the **`<microgroom>`**

flag to automatically manage the overall use of space in the statistics files. To “train out of class” texts (that is, texts that didn't score adequately badly in classes other than their correct class) use the **`<refute>`** flag (for the default build of Winnow where the promotion and demotion factors are approximately reciprocal values, this is approximately equivalent to untraining a mistaken training as well).

Hyperspace Classification

Hyperspace classification is a very different kind of classifier. Instead of grouping all members of a class of texts together and looking for common features, hyperspace classification keeps each text separate. Each text, known or unknown, represents a single point in a multidimensional hyperspace of roughly 4 billion dimensions. The classifier measures the distance between the point representing the unknown text and the points representing each of the known texts. This distance is calculated as the square root of the square of the number of dimensions that appeared in both documents divided by the product of the dimensions appearing in only one or the other texts (but not exactly; there is a modifier to limit the impact of texts that are extremely close together.) If **`<unique>`** is not specified, then each repeated feature counts as an additional unit displacement in that particular dimension of the hyperspace, otherwise only the first occurrence is counted (and so the hyperspace is restricted to the vertices of a 4-billion-dimensional hypercube).

Every known text generates one such distance to the unknown text (thus, a single document yields only one hyperspace feature, instead of the hundreds or thousands of features in the Markovian, OSB, or Winnow classifiers). The distances are then merged with a combined radiance model. An astronomical analogy is perhaps the best way to understand this- each known document is a star, shining in hyperspace. The position of each star depends on what feature phrases the document contained, and the brightness of the star depends on the number of feature phrases that the known document shares with the incoming unknown document.

Each set of documents (the good examples, or the spam examples) forms a galaxy of these stars, shining in the four-billion-dimension hyperspace. To determine which class our unknown document belongs, we just see which galaxy shines more brightly on the star of our unknown document. If the “good galaxy” shines more light onto our unknown text’s position in hyperspace, the unknown text is assumed “good”, and if the “spam galaxy” shines more light, then the unknown is assumed to be “spam”.

This sounds like a lot of work, but hyperspace classification is surprisingly fast (faster than any other classifier by 4 or more times), as accurate as any other classifier except OSBF, and uses very little disk space (less than any other classifier, by as much as a factor of 40!). Depending on the corpus, and the training method, hyperspace can sometimes even beat OSBF in accuracy. At this writing, the classifier is considered experimental only because the on-disk format of the hyperspace point sets is likely to change to use a little more space to get more speed and accuracy.

An advantage of the hyperspace classifier is that convergence is guaranteed; in fact, it's rarely even necessary to train more than once. This is because hyperspatial classification is not a linear classifier. For example, a hyperspace classifier can learn an "unlearnable" (to a linear classifier) checkerboard in O(n) examples, where N is the number of distinct squares on the checkerboard. In classical machine learning theory, the hyperspace classifier is a KNN (K-Nearest-Neighbor) classifier with N set to the entire trained corpus set and the proximity function equivalent to a modified Gaussian weighting.

The hyperspace classifier is activated with the **`<hyperspace>`** flag. You can use the **`<unique>`** flag with it, although the effect is not very large, it does decrease the disk space requirement even further. The hyperspace algorithm uses the OSB feature set, so using the **`<unigram>`** feature set flag works, but cuts accuracy significantly (and is therefore not recommended). As a somewhat experimental classifier, **`<microgroom>`** is not yet implemented, and **`<refute>`** is ignored.

Hyperspace classification is best trained SSTTT, with a very thin margin (only 0.25 to 0.5 units of pR needed; this means only errors and texts with nearly completely ambiguous results need to be trained. Hyperspace does work with pure TOE training, but loses significant accuracy (it does keep its speed and disk-space advantages).

Correlative Classification

Correlative classification is another very different classifier – it uses individual letters as features, not words or phrases. Because of this, you don't need to specify a word-parsing regex to a correlative LEARN or CLASSIFY – and if you do specify one, it will be ignored. Correlative classification accumulates points based on the run lengths of strings found in the unknown text that exactly match strings found in a known text. Strings with longer run lengths get more points and the corpus with the biggest point score at the end wins.

By and large, the correlative classifier is not a very good text classifier; it's slow and not very accurate compared to Winnow or OSBF. However, it can match binary data, images, executable files, and even highly compressed data. It should be considered a specialty tool, not for common use. If you need to use Correlate, train it with TOE.

The correlative classifier is activated with the **`<correlate>`** flag, and ignores **`<unique>`** mode, even if specified. Likewise **`<microgroom>`** and **`<refute>`** are also ignored. To undo a mistaken training with the correlative classification or discard no-longer-useful data, just edit the statistics file with a text editor and delete the undesired text sections.

Bit Entropy Classification

The Entropy classifier is another experimental classifier in CRM114. Instead of a Markov Random Field model (such as OSB, OSBF, or Markov classifiers), it uses only the Markov chains actually seen in the training corpus, and rather than measuring probability, it measures the number of bits of entropy in the unknown text, given the prior knowledge of the likely transitions in a given training corpus. The lower the entropy, the more "alike" the unknown text is to the prior knowledge in a particular training corpus. Entropy-based classification is activated with the **`<entropy>`** flag in a CLASSIFY or LEARN statement.

Entropy-based classification is equivalent to solving the first half of the "optimal lossless prior-only compression" problem – of finding the absolutely best way to losslessly compress an arbitrary file, given some prior knowledge in the form of files that are supposedly similar to act as background information, but not being allowed to adaptively alter the prior-knowledge statistics by what's found in the new file. Admittedly, compressors like gzip and LZW are allowed to use in-the-file information to adapt; however, we aren't interested in that. We care only how well the known text corpus can provide a compression model for the unknown text.

In fact, we don't even bother to actually create this best possible losslessly compressed text; we only care about how many bits it takes to express that compressed form, so that's all the Entropy classifier calculates (fractional bits are allowed).

The original inspiration for this classifier came from Andrej Bratko's[20] DMC filter shown at NIST TREC 2005[21]; however the code here is not related to Andrej's code, nor to Gordon Cormack's published DMC code; the algorithms do not currently implement Gordon Cormack's node-cloning, nor Matthew Young-Lai's node merging[22] (which is limited to re-merging nodes that were previously cloned from a single original.). Instead, different techniques are used to similar ends. Further, this classifier works best with open-ended Markov chains in the model, versus the toroidal lattices required by Cormack's DMC.

20 Bratko, Andrej, Markov Modeling for Spam Filtering NIST TREC 2005
21 Cormack, Gordon V. et al, Data Compression using Dynamic Markov Modeling (Computer Journal #6, 1987)
22 Matthew Young-Lai, Adding state merging to the DMC data compression algorithm, Information Processing Letters 70 (1999) 223–228.

Understanding entropy classification

Unlike most of the other classifiers, the default token of input text for the Entropy classifier is the bit; this classifier operates one bit at a time. Changing the possible token set (called the “alphabet” in information theory) can be done but requires a recompile and is not recommended. As such, no tokenization regex is needed.

To understand more deeply how the entropy classifier works, we need to learn just a little information theory, and in particular, the concept of “entropy”. Entropy is a way to measure the “uncertainty” of a system. More entropy translates to phrases like “more uncertain, given context”, “more random, harder to predict”, and “past performance is no gaurantee of future results”, and is measured in bits. These are the same “bits” as you normally think of in a computer, and the equivalence is related to compression- how many bits does it take to optimally compress the input, given that you’re allowed to use any trick you can think up[23].

Here are some examples of how entropy scales with uncertainty. Something completely unpredictable (like flipping of a fair coin) yield very high entropy rates (and indeed, a fair coin yields exactly 1.00000 bits of entropy per flip). Something a little more predictable, such as a “trick” coin which has had the coin edges filed to give 55 % heads and 45 % tails yields a little less entropy (a 0.55 / 0.45 coin gives 0.9927745 bits per flip). The carnival ring-toss game where you stand one chance in 100 of getting the ring to stay on the post has an entropy of just 0.08079 bits per toss (because it’s fairly predictable you’ll miss) and the roughly one-in-a-million chance per year of being hit by lightning is 0.00002 bits per year (it’s rather predictable that you won’t be hit by lightning in any given year).

Entropy can be generalized to more than a yes/no bit. Some sources have many different symbols in their alphabet of possible outputs; it’s necessary to sum up the average entropy of all of the possible outputs to get the actual entropy of the signal source. Shannon’s equation does this summation over all of the possible symbols that a source can send (including the “send no symbol at all” symbol[24]); here’s the mathematical definition:

$$\text{Entropy} = \sum_{\text{all possible symbols from this source}} [\, - \text{Prob}_{\text{symbol } \mathbf{x}} * \log2 (\ \text{Prob}_{\text{symbol } \mathbf{x}}\ )\,]$$

The derivation of this equation can be handwaved as follows: the best possible encoding of any symbol encodes that symbol using log2 bits of the probability of that signal (Huffman’s theorem); symbols we are very likely to see are coded with very few bits (we

23 Optimal compression is also related to Kolmogorov complexity – the smallest program that can reproduce the input.

24 Consider this as the embodiment of “and if you choose not to decide, you still have made a choice”.

see them all the time, so optimally we should not spend a lot of bits on them), while unlikely symbols are coded with a large number of bits (they occur rarely, so the impact of a lot of bits doesn't happen often). We then take the average entropy contributed by this symbol to the source, compared to the entire alphabet of the source; that's the probability of the symbol times the log2 of the probability. Finally, we add up the entropy contributions over all possible symbols from this source; that's the total entropy on average of this source.

The Entropy Classifier Internals

In order to use entropy classification, we have to build an internal model of the source we <u>think</u> is emitting the stream of symbols that we're classifying. In the general case, this is a Markov chain; there are a number of ways to build the Markov model, including a provably optimal method[25], Because we want to support fast incremental learning even if the final model is somewhat suboptimal, we use one of two "on the fly" methods – starting with a large, predetermined, very general structure and only varying the per-node transition probabilities, or starting with a very simple structure and adding nodes and transitions as they are encountered in the training corpus.

Entropy Classifier Initialization and Learning

The first (and default) method starts out with a completely static layout of the Markov model; it's a "perfect shuffle" toroidal graph composed of 256 rows and 256 columns[26]; each node is preconnected to two nodes in the next column as the "next symbol is 0" and "next symbol is 1" Markov transitions. Because the graph is a perfect shuffle, the first node in each column then has two incoming 0-transitions, and the second node in each column has two incoming 1-transitions. The START node is column zero, row zero, but this is actually immaterial because the toroidal graph is rotationally symmetric in both rows and columns.

This toroidal graph results in a Markov model tht has no "edge" - no matter what sequence of 0s or 1s comes in, the model always has a "next node" link available. It also has relatively few nodes allocated (by default, just 64Knodes) and unless node splitting is enabled, the memory requirement does not grow (this is handy for embedded systems). That's the good news. The bad news is that the model starts out containing nodes with no

25 There is a published algorithm (the Baum-Welch algorithm) to generate an optimal Markov model in $O(n^2)$ operations. However, our N is on the order of 10 to the eighth bits, so we'd need on the order of 10 to the sixteenth cycles to build an optimal model.

26 The number of rows and columns are set at compile time since they don't have a big impact on the accuracy,

prior information, and so whenever an incoming text goes off-track compared to a previously seen node, then there's no prior information and the entropy jumps to about 1 bit of output per bit of input until we stumble all the way around on the toroidal graph and re-encounter a node that has nonzero training.

The second mode of initialization uses uniquely reachable nodes (set by the **`<unique>`** flag) and initializes only a single START node and its two decendants as parts of the graph, and then allocates a relatively large freelist (default is one million nodes). As we LEARN, we walk the graph of previously seen bits, incrementing the visits on each transition until we hit a transition that has never been encountered before. At this point, we allocate a new node from the free list and link it in, and proceed to the next bit. This allocation continues until we run out of input – or out of free nodes. Each node is thus reachable only by a unique prefix sequence.

In order to minimize the consumption of free nodes, the entropic classifier has an optional minimization strategy called **`<crosslink>`**. Crosslinking[27] is exactly what it sounds like – when we fall off the edge of the graph while LEARNing, rather than allocate a new node, we first check the already-extant graph for a node "sufficiently alike" what we are about to allocate, and if it exists, we just create a link crossing over to that "sufficiently alike" node. This search is actually very fast; the CRM114 entropic classifier uses an FIR model[28] of the prior transitions and a lookaside table to find the "most alike" node (in an average of three node examinations).

During classification, the procedure is the same for both the default toroidal and unique-state models; walk the graph, adding up the actual number of fractional bits that an optimal compression technology would have required to losslessly compress the input file. The number of bits required is calculated as:

$$\begin{aligned} \text{Bits\_required} &= -\log2\ (\text{Prior\_pobability}) \\ &= -\log2\ (N_{\text{seen\_this\_bit\_here}} / N_{\text{total\_bits\_here}}) \end{aligned}$$

(with a small "nerf factor" added to prevent exactly-zero issues in both the division and the log). The quality of match is the total of the number of bits required to optimally do a lossless compression of the unknown input file, and smaller is better. We should remember that the Entropy classifier doesn't actually <u>construct</u> that perfect lossless compression bit sequence, but merely calculates the length of that perfect compression.

27 Crosslinking is named after the reaction in polymer chemistry that forms bonds between two already-completed chains of molecules, and is responsible for things like durable rubber treaded tires.

28 done by using an arithmetical coding of the prior bit history entering each node; more recent bits in the history have higher significance in the arithmetical code.

If the model is the default toroidal model, there's no chance of running off the edge of the model (every node already has a subsequent-0 node and a subsequent-1 node preallocated), but the **`<unique>`** models have no such gaurantee. Inside a **`<unique>`** model, if we hit a no-subsequent-node situation during classification, we have no choice but to find a similar node in the already-existing model and transition there[29]. The actual entropy "hit" for such a crosslink-like jump is fairly small, since both the virtual transmitter and virtual receiver can perform the same calculation and decide what the destination node will be, and the same FIR prior and lookaside table structures are available for node lookup accelleration.

Additionally, such crosslink-type jumping can be enabled for <u>any</u> node where there is insufficient prior experience to calculate the entropy needed to go to the next state; this is enabled by the **`<crosslink>`** flag; this is flag is recommended for both the default toroidal models and the **`<unique>`** models.

The **`<refute>`** flag is partially supported; the node transition counts are decremented, but (currently) the nodes with zero counts aren't returned to the free pool; this is a benign bug unless you make lots and lots of mistakes in training or you use DSTTT, so DSTTT is <u>not recommended</u> for entropy classification. [It's also problematic in DSTTT that if you "untrain" something you never trained, what do you do when you run off the edge of a **`<unique>`** model? You can't allocate a node and give it -1 probability of transitioning.]

Overall, the Entropy classifier tends to be slower than OSB, OSBF, or Winnow (on a par with Markov classification), and about as accurate than all of these, when run with the recommended **`< entropy unique crosslink >`** flags for both LEARNing and CLASSIFYing. Entropic classification is usually a little less accurate than OSB, OSBF, Winnow, or Hyperspace when the cost of misclassification is equal in either direction, but it has lower total error costs when the cost of one or the other kind of error are wildly different.

Entropy classification works best using the default 5E-7 crosslinking threshold and SSTTT trained with 0.25 pR thickness. To override the crosslink threshold, specify the value as the first /slashed value/. A value of zero effectively turns off crosslinking even if **`<crosslink>`** is specified, and is not recommended for good performance. Other flags such as **`<unigram>`** and **`<microgroom>`** are not supported by the entropic classifier and will be ignored without issuing an error message. Because the file format of the entropy classifier is likely to change in the near future, it should be considered "experimental" and not used for production deployment unless the risks are acceptable.

29 The other obvious choice- to branch back to the START node – yielded very poor accuracy in testing and isn't available without editing the source code and recompiling.

Summary of the Standard CRM114 Classifiers

Classifier	*Use on*	*Recommended Flags*	*Recommended Training*	*Comments*
Markovian (default)	Human text	microgroom	TOE or SSTTT, pR 20.0	Obsolete – use OSB or OSBF (Markovioan is retained only for back compatibility)
OSB	Human text	osb unique microgroom	SSTTT pR 10.0, DSTTTR, pR 10.0	A good standard, well tested
OSBF	Human text	osbf unique microgroom	DSTTTR, pR 10.0	A good standard, also well tested
Unigram (standard Bayesian text classifier)	Human text	osb unigram	DSTTTR, pR 10.0	Only use for comparisons against Bayesians; NOT as good as OSB/OSBF
Winnow (Lightstone)	Human text	winnow unique microgroom	DSTTT only, with pR 20.0	Non-probabalistic; neural-net-like.
Hyperspace (infinite range K-nearest-neighbor)	Human text, token sreams	hyperspace -or- hyperspace unique	TOE or SSTTT, pR 0.5	Very fast learner, very small disk footprint. **Semi-experimental.**
Correlator (run length weighted char matching)	Character and compressed data	correlate	TOE only	Special purpose only- poor speed performance on plaintext.
Entropy (lossless compression)	Text and lightly compressed data (.GIF, .JPG, .MP3)	entropy unique crosslink	SSTTT, pR 0.25	Less accurate than other classifiers when P(false neg) ~ P(false positive). **Experimental**

Why pR ?

The CRM114 classifiers generally report their results in a scale called pR, not probability.

The pR scale was originally designed because many of CRM114's pseudo-Bayesian classifiers report statistical results with fifty or even a hundred 9's of certainty; clearly those high probabilities are artifacts of using the Naive Bayesian combining rule on data with significant correlation.

The pR value of a classification is defined for Markovian matching as

$$\log_{10}\text{ (in-class probability)} - \log_{10(}\text{ (not-in-class probability)}.$$

Typically, this gives a result in the range of +320 to -320 for IEEE floating point input probabilities. This puts classification confidence into a much more human scale – and allows the classifier designer some degrees of latitude in expressing the outputs of new classifiers.

CRM114 classifiers are generally set up to give pR values in this scale; experimental classifiers may need to be adjusted to get their typical results into the following ranges:

- pR values above +100 signify "high chance this text is a member of this class"
- pR values between +100 and +10 signify "moderate chance this text is a member of this class"
- pR values between +10 and -10 signify "unsure" (and typically should be retrained if using either SSTTT or DSTTT training.
- pR values between -10 and -100 signify "moderate chance this text is not a member of this class"
- pR values of below -100 signify "high chance this text is not a member of this class"

Experimental and semi-experimental classifiers will often still need to have their pR calculations "tweaked". That's one risk of running a non-production classifier.

Examples of Learning and Classification

Here's some simple code to classify the incoming text into one of three classes A, B, or C. Note that we don't have any preconceived notion as to what is an "A", "B", or "C" text are; we need to learn those as we go.

In this example, we'll implement this as two programs – the first is only a classifier program and prints out which class we think the input text is. The second program is the actual learning program and we will manually invoke it whenever the classifier makes a mistake, telling it which of the three statistics files we want to learn "into". We will use TOE learning (Train On Error) only, and the OSB classifier. We will also use the **`<unique>`** modifier, as well as **`<microgroom>`** to automatically manage space in our statistics files.

To make the code simpler, we'll name our class statistics files **A, B** and **C**, with no extension. But be warned; the recommended default for statistics files is to use an extension of .css for Markovian and OSB files, .cfc for OSBF files, .cwi for Winnow files, .chs for Hyperspace files, and .cor for correlation files. CRM114 does not enforce these filename extensions in any way, but if you use a different filename extension, you run the risk of confusing everyone who follows you, so document your aberrational design carefully.

For the sake of this example, we won't do any interpretation of the **:stats:** variable, we'll just print it out so we can see what it looks like (of course, we can MATCH inside it if we wanted). We also don't use the **|** separator, so all three files are "success" files and this particular CLASSIFY statement will never FAIL to the end of the **{}** block. We'll also take the default word tokenizing regex which is **[[:graph:]]+** (that is, a classification word is any sequence of printing characters).

We'll also use the default CRM114 behavior of "read standard input till EOF before starting program execution", as this is exactly what we would want to do in any case. We will use the OSB classifier, with both **<unique>** and **<microgroom>** enabled. Here's the code:

```
{
      isolate (:stats:)
      classify <osb unique microgroom>  (A B C)  (:stats:)
      output /:*:stats:/
}
```

The learning program is even simpler, just one line of code. We will use **:_arg2:** to tell us which of our three statistics files we want to learn into, but otherwise, we're identical to the classification program above (default word tokenizer of **[[:graph:]]+**, default of "read standard input till EOF", etc). Here's the code:

```
{
      learn <osb unique microgroom> (:*:_arg2:)
}
```

One last detail – we need to create our statistics files before we can use them to CLASSIFY. We can use the `cssutil` utility program to do this, but we can also create them from within CRM114, just by doing a LEARN on an empty data input. (That's a handy programming trick – make a mental note that LEARN will create a new empty statistics file the desired one does not yet exist. You can use cssutil to generate empty .css files, and osbf-util to generate the .cfc files, but those are special-case programs. The always-works way is to use a LEARN on a very short example text to create the new, properly formatted statistics file.)

```
# crm learn1.crm A
hit ctrl-D
# crm learn1.crm B
hit ctrl-D
# crm learn1.crm C
hit ctrl-D
# ls -la A B C
-rw-r--r--  1 wsy wsy 12582924 Jan 26 07:13 A
-rw-r--r--  1 wsy wsy 12582924 Jan 26 07:13 B
-rw-r--r--  1 wsy wsy 12582924 Jan 26 07:13 C
#
```

Notice that the statistics files A, B, and C are rather large – 12 megabytes. That's the default size for a statistics file, but you can often get good classification with much less. (use the -s or -S flag to specify just how big you want your statistics files. These flags only set the slotcount used when creating a new file; they don't affect file sizes once the file has been created.)

Now, let's actually classify a text. Note that we haven't actually learned anything yet, so our "classification" is going to be completely vacuous. Here we go:

```
# crm classify1.crm
the quick brown fox jumped over the lazy dog's back
CLASSIFY succeeds; success probability: 1.0000  pR: 304.6527
Best match to file #0 (A) prob: 0.3333  pR: -0.3010
Total features in input file: 40
#0 (A): features: 1, hits: 0, prob: 3.33e-01, pR:  -0.30
#1 (B): features: 1, hits: 0, prob: 3.33e-01, pR:  -0.30
#2 (C): features: 1, hits: 0, prob: 3.33e-01, pR:  -0.30
#
```

Remember, there's no real data here yet- the A, B, and C files are all empty. So, each statistics file is an equally good (or bad) match, with probability 0.333333 . Also, because all three files are considered "success" files, the CLASSIFY is an overall success (and doesn't FAIL to the end of the `{}` block).

Now, let's train in a few pieces of text. A quick perusal of Project Gutenberg's[30] E-texts shows that the works of Shakespeare, Lewis Carroll, and Sir Arthur Conan Doyle are all in the public domain; further, Carrol and Doyle are nearly contemporaneous. We'll use the following texts as training and unknown texts (mnemonic: A is for Arthur, B is for Bill, and C is for Carroll). Because Doyle's <u>The Hound of the Baskervilles</u> is so much longer than

30 You can reach (and support) Project Gutenberg at **`http://www.gutenberg.org/`**

the other texts, we'll use only the first 500 lines of it.:

Statistics File	*Author*	*Example Text*	*Unknown Text*
A	Sir Arthur Conan Doyle	The Hound of the Baskervilles (classic Sherlock Holmes, first 500 lines)	The Poison Belt (a Professor Challenger story, in "The Lost World" universe)
B	William Shakespeare	Hamlet, Act V, Scene I (Yorick's skull being exhumed from the cemetery, all 679 lines)	Macbeth, Act IV, Scene I (the scene in the cave with the witches and the cauldron)
C	Lewis Carroll	Through the Looking Glass, chapter 1 (The Looking Glass House, 133 lines)	Alice in Wonderland, chapter 1 (Down the Rabbit Hole)

Now, we'll train in the A, B, and C example texts:

```
# crm learn1.crm A < Hound_of_the_Baskervilles_500.txt
# crm learn1.crm B < Macbeth_Act_IV_Scene_I.txt
# crm learn1.crm C < Through_The_Looking_Glass_Looking_Glass_House.txt
```

(this took on the order of two seconds total CPU time on a sub-GHz Transmeta laptop.)

We can now test our "quick brown fox" text again. It shouldn't match any of the texts well, but the results should not be an absolute equi-probable match:

```
# crm classify1.crm
the quick brown fox jumped over the lazy dog's back
return, then ctrl-D
CLASSIFY succeeds; success probability: 1.0000  pR: 304.6527
Best match to file #0 (A) prob: 0.3634  pR: -0.2435
Total features in input file: 40
#0 (A): features: 14394, hits: 36, prob: 3.63e-01, pR:  -0.24
#1 (B): features: 9465, hits: 31, prob: 3.17e-01, pR:  -0.33
#2 (C): features: 11796, hits: 17, prob: 3.20e-01, pR:  -0.33
#
```

showing that none of the documents matches very well, but the best match is to file A (Arthur Conan Doyle) which isn't all that surprising, as Hound of the Baskervilles certainly alludes to chasing the foxy criminal element. Let's continue, feeding in short chunks of each of The Poison Belt, Hamlet, and Alice in Wonderland.

First, we'll try classifying 22 lines from from The Poison Belt by Sir Arthur Conan Doyle (who also wrote Hound of the Baskervilles). The Poison Belt is not a Sherlock Holmes story, so there's no mention of the Holmesian environment and no Baker Street cliches:

```
# crm classify1.crm
This was the letter which I read to the news editor of the
Gazette:--


"SCIENTIFIC POSSIBILITIES"

"Sir,--I have read with amusement, not wholly unmixed with some
less complimentary emotion, the complacent and wholly fatuous
letter of James Wilson MacPhail which has lately appeared in
your columns upon the subject of the blurring of Fraunhofer's
lines in the spectra both of the planets and of the fixed stars.
He dismisses the matter as of no significance.  To a wider
intelligence it may well seem of very great possible
importance--so great as to involve the ultimate welfare of every
man, woman, and child upon this planet.  I can hardly hope, by
the use of scientific language, to convey any sense of my
meaning to those ineffectual people who gather their ideas from
the columns of a daily newspaper.  I will endeavour, therefore,
to
condescend to their limitation and to indicate the situation by
the use of a homely analogy which will be within the limits of
the intelligence of your readers."
return, then ctrl-D
CLASSIFY succeeds; success probability: 1.0000  pR: 304.6527
Best match to file #0 (A) prob: 0.9994  pR: 3.1921
Total features in input file: 676
#0 (A): features: 14394, hits: 1829, prob: 9.99e-01, pR:   3.19
#1 (B): features: 9465, hits: 1199, prob: 4.97e-04, pR:  -3.30
#2 (C): features: 11796, hits: 638, prob: 1.45e-04, pR:  -3.84
#
```

As one might hope, file A (for "Arthur Conan Doyle) is the best match for this text, with a probability of 0.9994 (to four significant digits) and a pR of 3.19 . This is a very encouraging result on 500 lines of known input and 20 lines of test data.

Let's try again, with a "B" file (B for our friend William "Bill" Shakespeare), with the witches stirring their brew:

```
# crm classify1.crm
First Witch

    Round about the cauldron go;
    In the poison'd entrails throw.
    Toad, that under cold stone
    Days and nights has thirty-one
    Swelter'd venom sleeping got,
    Boil thou first i' the charmed pot.

ALL

    Double, double toil and trouble;
    Fire burn, and cauldron bubble.
return, then ctrl-D
CLASSIFY succeeds; success probability: 1.0000  pR: 304.6527
Best match to file #1 (B) prob: 0.4535  pR: -0.0809
Total features in input file: 176
#0 (A): features: 14394, hits: 59, prob: 3.43e-01, pR:  -0.28
#1 (B): features: 9465, hits: 81, prob: 4.54e-01, pR:  -0.08
#2 (C): features: 11796, hits: 4, prob: 2.03e-01, pR:  -0.59
#
```

showing that although this ten-line sample is not quite enough for certainty, the result is that this text was most likely penned by the same author as The Scottish Play. This is certainly an acceptable result with only 400 lines of training text and 10 lines of sample.

Finally, let's see how Lewis Carroll fares; our sample text (chapter 1 of Through The Looking Glass) is only 133 lines long and our testing text just four lines, a little too short for significant statistics to accumulate:

```
# crm classify1.crm
The rabbit-hole went straight on like a tunnel for some way, and
then dipped suddenly down, so suddenly that Alice had not a moment
to think about stopping herself before she found herself falling
down what seemed to be a very deep well.

CLASSIFY succeeds; success probability: 1.0000  pR: 304.6527
Best match to file #0 (A) prob: 0.7119  pR: 0.3929
Total features in input file: 172
#0 (A): features: 14394, hits: 411, prob: 7.12e-01, pR:   0.39
#1 (B): features: 9465, hits: 209, prob: 4.73e-02, pR:  -1.30
#2 (C): features: 11796, hits: 236, prob: 2.41e-01, pR:  -0.50
#
```

which is fairly reasonable considering the paucity of our example text- just 133 lines of sample and *only four lines* of unknown text..

Words of Wisdom from the Voice of Experience:

In the above examples, we clipped each of the example training texts to be roughly the same length (around 500 lines). If you repeat the examples above, but with the full Hound of the Baskervilles text (all 7235 lines), you'll find that *all* of the unknown texts appear to come from Hound of the Baskervilles. This is because we're not training on each section of Hound individually but rather forcing the entire text into the statistics file in one large chunk.

Normally you'll be training with a large number of texts (preferably using TOE, TOER, or TUNE), so this won't be an issue, but beware of situations (or project managers) that demand you train in a single large text, or train in an entire corpus as a single large block. If you train a single huge block of text, you will almost certainly get inferior accuracy. Instead, you should break the sample text up into smaller sections, so that you have at least a few dozen, if not a few hundred separate samples of text, and then use TOE, TOER, or TUNE on those samples as if they were independent texts.

Utilities for Manipulating Statistics Files

Besides using LEARN and CLASSIFY to operate on statistics files, we can also use some specialized standalone utilities. Not every utility is supported on every statistics file type; there are some glaring omissions. Further, not every type of statistics file is self-identifying, so some utilities will think they are operating correctly on a file of a type they understand, but in reality the statistics file belongs to a different classifier entirely and the utility program is operating in a totally deranged manner.

CSSUTIL and OSBF-UTIL – basic statistics file information.

For Markovian and OSB files, the **cssutil** program allows command-line statistics detailing and some useful manipulation of the data storage. The exact options will vary depending on the version of CRM114 you are running, but **-h** will get you a list of capabilities in your copy:

```
# cssutil -h
cssutil version 20050801-BlameTomotoyo - generic css file utility.
Usage: cssutil [options]... css-file
                -b   - brief; print only summary
                -h   - print this help
```

```
                    -q   - quiet mode; no warning messages
                    -r   - report then exit (no menu)
                    -s css-size  - if no css file found, create new
                                   one with this many buckets.
                    -S css-size  - same as -s, but round up to next
                                   2^n + 1 boundary.
                    -v   - print version and exit
                    -D   - dump css file to stdout in CSV format.
                    -R csv-file  - create and restore css from CSV
#
```

Note that there is no option to retrieve the actual texts used to generate this particular statistics file. That's because most of CRM114's classifiers use a fast hashing method to both compress and store the data; that means there's no text stored. What's worse is that the hash isn't easily invertible (although it's not a "salted hash" so exhaustive enumeration would reveal the feature texts). This is all for the sake of classification speed, but it has the additional benefit of providing privacy against casual snooping of your training data.

The most useful output of **cssutil** is the percent utilization of any particular statistics file. This can be checked if it seems that the classifier programs are slowing down or that accuracy is being lost.

Because OSBF uses a slightly different storage layout, OSBF statistics files use the **osbf-util** utility, which works nearly identically to the Markovian/OSB **cssutil** utility.

Checking statistics differences with cssdiff

Because of the internal structure of a statistics file is not text, you can't use **diff** to see the differences between two statistics files (even if they are from the same classifier family). Instead, you need to use a specialized tool to compare the features as hashed. That tool is cssdiff (and works only on Markovian/OSB statistics files). The command format is just:

```
# cssdiff
Usage: cssdiff <cssfile1> <cssfile2>
#
```

where <cssfile1> and <cssfile2> are the two statistics filenames (without the < > brackets). The output is a summary of how similar the files are, and how different the files are. Note that this utility exists only for Markovian/OSB statistics files, and will not work correctly on any other statistics file type.

Merging statistics files with cssmerge

Sometimes it's necessary to merge two statistics files- or more often, to merge the contents of a small and nearly overfull statistics file into a new, much larger file. Simply copying the small file into the start of the larger won't work, nor will appending the smaller to the larger (or vice versa); this is because the internal structure of most CRM114 statistics files use embedded byte offsets and those offsets must be correct for any particular data entry to actually be found.

To merge a statistics file's data onto another statistics file, use the special utility, **`cssmerge.`** This utility is only available for Markovian/OSB statistics files. The command format is:

```
# cssmerge
Usage: cssmerge <out-cssfile> <in-cssfile> [-v] [-s]
 <out-cssfile> will be created if it doesn't exist.
 <in-cssfile> must already exist.
  -v           -verbose reporting
  -s NNNN      -new file length, if needed
#
```

Note, that the input file must always exist, but the output file will be created (and initialized to an empty state) if it isn't already in existence.

The big use of **`cssmerge`** is when a .css file has become too full and rather than using microgrooming (which loses information), it's desired to make the .css file bigger. Typically, one would do something like this:

```
# ls -la *.css
-rw-r--r--  1 root root 12582924 Sep  6 08:07 nonspam.css
-rw-r--r--  1 root root 12582924 Sep  6 08:07 spam.css
# mv spam.css oldspam.css
# cssmerge newspam.css oldspam.css -s 2000000

Overriding new-create length to 2000000

Creating new output .CSS file newspam.css

Output sparse spectra file newspam.css has 2000000 bins total

Input sparse spectra file oldspam.css has 1048577 bins total
# mv newspam.css spam.css
#
```

This copies all of the data in the original .css file into a new .css file, but the new .css file has 2 million (decimal) bins, versus the old .css file with only 1 megabins.

Note also that we haven't deleted the old .css file; this will serve us as a backup copy in case we do something very wrong.

Catching errors with FAULT and TRAP

Many programming languages support the concept of a programmer-supplied error handler, and CRM114 does too. With very few exceptions, all CRM114 execution engine faults, user-programmed faults, and several preprocessing faults can all be caught by user-supplied routines. The error-trapping statement is TRAP, and you can test it or use it for your own errors with the error-synthesizing statement FAULT.

Here's the syntax of FAULT and TRAP:

```
fault /var-expanded fault string/

trap /trapping_regex/ (:optional_fault_string_var:)
```

Note that the FAULT must supply a string; this string is passed on to the TRAP statement for two purposes:

1. the TRAP regex must match inside the FAULT string,
2. if the TRAP has a **`:fault_string_var:`** then the the fault string var is isolated and gets the value of the fault string. This is how a TRAP can tell why it was called.

In CRM114, error trapping is scoped *lexically*, that is, according to the curly braces **`{`** and **`}`** . The upside of this is that it's easy to error-trap a particular section of code; the downside is that your fault traps don't "follow" you down into subroutine calls. This is the reverse of many languages, where error trapping is "outside" of program lexical scope, but variables are "inside" lexical scope. This is actually a good thing, as it means that your error handler will never be called upon to service an error that that the error-handler's author never expected.

As in most other languages, CRM114 error traps nest, and each trap gets a chance to service the fault (to end it), service the fault part way (and pass it on), or simply pass on the fault unchanged.

The best news of all is that instead of requiring enumeration of every possible program fault condition, and setting independent traps for each one, CRM114 allows the programmer to use a regex at the TRAP level to determine if the trap is capable of handling the problem. If the regex matches the error message, the TRAP runs; if not, the next trap outward (in lexical ordering) is tested for regex match, and so on. Only if the fault gets outside of all user-supplied traps does the CRM114 default handler run (usually printing out a nasty message). If the error was a FATAL error, and no user trap was supplied, program execution ends. If the error was only a WARNING, up to 100 of them can occur unTRAPped before program execution is terminated (the 100 is a compile time parameter; change it if you want). There is no limit on the number of successfully trapped FATAL and WARNINGs that can be trapped, so feel free to use FAIL and TRAP as a programmatic construct, if you like that kind of programming style.

Here's an example of a TRAP in action. First, we'll write a simple program that gets an error (in this case, it tries to INPUT from a nonexistent file):

```
{
      window
      input  [nonexistent_file.txt ]
}
```

which gets the following error:

```
# crm trap1.crm

crm: *ERROR*
 For some reason, I was unable to read-open the file named
nonexistent_file.txt
 Sorry, but this program is very sick and probably should be killed
off.
This happened at line 3 of file trap1.crm
The line was:
--> input  [nonexistent_file.txt ]

#
```

This is all well and good, except that your program might prefer to execute some fix-up routine when this file isn't available. We can set a TRAP for that eventuality (here, it's just to print out an error message, but you can do any fix up you want, including SYSCALLing a program that might be able to create the missing file):

```
{
      window
      input  [nonexistent_file.txt ]
      #
      #     Here's the trap code
      trap /read-open/ (:errmsg:)
      output /\n---My program got an error on read-opening. --\n/
}
```

Now when the error occurs, the program will jump to the TRAP statement, and try to match the error message against the TRAP's regex (here, it's **`/read-open/`**). If the regex matches, then the program execution continues from the TRAP statement (and the text of the error message is captured as the ISOLATEd variable named in the TRAP statement). If the regex didn't match, then the next TRAP statement in line gets to try its regex; when there are no more TRAP statements, the default error handler prints out the error message and terminates the program. Let's give it a run:

```
# crm trap2.crm

---My program got an error on read-opening. --
#
```

which shows that the TRAP worked as expected.

Now, what if the file did exist? Wouldn't the TRAP code get executed anyway? No, because a TRAP is like an ALIUS statement – if you got to a TRAP “normally”, (that is, not as the result of an error or a FAULT statement) the TRAP skips to the end of the current **`{}`** block. Let's test it:

```
{
      window
      input  [ /etc/fstab ]
      accept
      #
      #     Here's the trap code
      trap /read-open/ (:errmsg:)
      output /\n---My program got an error on read-opening. --\n/
}
```

The only change is that we've changed the nonexistent text file to `/etc/fstab`, which will be found, and added an ACCEPT statement to print it out. (programming hint – the '/' characters in the filename `/etc/fstab` do NOT need to be escaped because the [ ] boxes are “outermost” quote characters for the string, and only the outermost quote characters matter in CRM114. If this had been an argument that was quoted with `'/'` characters, we would have needed **`\`**-escapes)

Here's the result (edited to fit on the page nicely):

```
# crm trap3.crm
# This file is edited by fstab-sync - see 'man fstab-sync' for
details
LABEL=/1      /                       ext3    defaults        1 1
```

```
LABEL=/        /RH7.3                  ext3    defaults        1 2
LABEL=/boot    /boot                   ext3    defaults        1 2
none           /dev/pts                devpts  gid=5,mode=620  0 0
none           /dev/shm                tmpfs   defaults        0 0
LABEL=/home    /home                   ext3    defaults        1 2
none           /proc                   proc    defaults        0 0
none           /sys                    sysfs   defaults        0 0
/dev/hda6      swap                    swap    defaults        0 0
#
```

Notice that our error message did not print out, showing that a TRAP is indeed a skip to the end of the current **{}** block if no error occurs.

User-Defined FAULTs

CRM114 allows the user to create any arbitrary fault as needed, by using the FAULT statement. The FAULT statement has one argument – a var-expanded string. This string becomes the equivalent of the error message, and any TRAP that wants to catch this user-defined FAULT must have a matching regex.

Here's an example; we'll FAULT with "No Cheshire Cat Seen" and see what happens if there's no TRAP to catch this FAULT:

```
{
      window
      output / Looking for a cheshire cat \n/
      fault / No Cheshire Cat Seen /
      output / --- this line should never execute --- /
}
```

here's the result:

```
# crm trap4.crm
 Looking for a cheshire cat

crm: *ERROR*
 Your program has no TRAP for the user defined fault:  No Cheshire
Cat Seen
 Sorry, but this program is very sick and probably should be killed
off.
This happened at line 4 of file trap4.crm
The line was:
--> fault / No Cheshire Cat Seen /

#
```

which is encouraging, as it shows that not only did we manage to create our own FAULT, but that in the event that we forget to create a corresponding TRAP that the CRM114 runtime system will catch the problem and report it as a fatal error.

If we add the proper TRAP statement, the code looks like this:

```
      {
            window
            output / Looking for a cheshire cat \n/
            fault / No Cheshire Cat Seen /
            output / --- this line should never execute --- /
            trap /.*/ (:my_error_message:)
            output /\n Got the trap for “:*:my_error_message:” \n/
      }
```

and we get the result:

```
      # crm trap5.crm
       Looking for a cheshire cat

       Got the trap for “ No Cheshire Cat Seen ”
      #
```

which is exactly as we would hope for.

Nesting TRAPs

We mentioned above that your TRAP statements will nest. This is done in two ways:

- If the trap `/regex/` doesn't match, the TRAP doesn't react at all to the fault, and the FAULT is tested against the next trap statement outward. If this is acceptable then the programmer doesn't need to do anything special as this is the default behavior.

- Even if the trap `/regex/` matches, it's possible the trap fix-up code will conclude that it cannot fix the problem, and will want to pass the fault upward.

The second case requires that the program have a FAULT inside the TRAP code to kick back into TRAP processing.

Here's a contrived example. We'll create two nested TRAPs, and the first (inner) TRAP will just print out a short message, showing that it really did get control, and pass the fault to the outer TRAP. The outer trap will print out another error message and then the program will exit

Here's the code, adapted from our examples above:

```
	{
		window
		output / Looking for a cheshire cat \n/
		fault / No Cheshire Cat Seen /
		#
		output / --- this line should never execute --- /
		#
		trap /.*/ (:my_error_message:)
		output /\n I'm the inner trap, msg: “:*:my_error_message:” \n/
		fault / Re-faulting on: :*:my_error_message: /
		#
		output / --- this line should never execute --- /
		#
		trap /.*/ (:outer_message:)
		output /\n I'm the outer trap, msg: “:*:outer_message:” \n/
	}
```

When run, we get:

```
	# crm trap6.crm
	 Looking for a cheshire cat

	 I'm the inner trap, msg: “ No Cheshire Cat Seen ”

	 I'm the outer trap, msg: “ Re-faulting on  No Cheshire Cat Seen  ”
	#
```

showing that we can nest TRAPs.

You might wonder why we didn't need to wrap the inner FAULT and TRAP in **{}** braces. The answer is that TRAPs are scoped not just lexically, but also in first-seen-first-tried sequence (again, this is the same style as ALIUS). Thus, a TRAP in an inner **{}** nesting won't trap a fault originating in an outer **{}** nesting level, but you can have any number of TRAPs at any desired nesting level without having to add extra **{}** nestings. This is especially handy because it keeps the code that might FAULT and the associated TRAP fix-up code close together.

The Mother Of All TRAPs

Good programming practice for system services dictates that you should attempt to at least gracefully error out on any program error, no matter how bad the error; printing an error message and setting a nonzero exit code is considered reasonable. CRM114 makes this particularly easy; just put a TRAP with the match-anything regex `/.*/` at the very end of your code, followed by an OUTPUT statement and then an EXIT with a nonzero exit. Here's an example, with this global TRAP giving exit code 99:

```
{
      window
      #
      #  ....  your whole program goes here ...
      #
}
trap /.*/ ( :my_global_trap_msg: )
output /\n ERROR:  this broke:   :*:my_global_trap_msg: \n/
exit /99/
```

To show that this code is harmless to programs that don't cause FAULTs:

```
# crm trap7.crm
#
```

Nothing happened; this is what we'd hope for. Now, let's throw in a FAULTing error:

```
{
      window
      #
      #  ....  your whole program goes here ...
      #
      fault / Some Bizarre Error Happens Here /
}
trap /.*/ ( :my_global_trap_msg: )
output /\n ERROR:  this broke:   :*:my_global_trap_msg: \n/
exit /99/
```

and we get:

```
# crm trap8.crm

 ERROR:  this broke:    Some Bizarre Error Happens Here
# echo $?
99
#
```

as desired (the bash command echo $? prints out the exit code of the last program executed) This is a good trick to remember when writing code that will be run by the system, like filters that will be run by procmail , as procmail is quite intent on using error codes to determine status of subsidiary processes.

Retrying the Failing Statement

If the TRAP was able to fix the problem that caused the FAULT, then it's possible for the TRAP routine to do a GOTO back to the failing statement, to retry the failing statement and reestablish the normal program execution path. To do this, the fix up routine needs to examine the fault string passed by the faulting statement, and use a MATCH to extract the line number of the failing statement.

All fault strings generated by the CRM114 runtime system will contain the text:

```
This happened at line N
```

with the line number of the failing statement as N. This allows an easy MATCH to extract the faulting line number. If a user-triggered FAULT will be subject to a GOTO-style retry, it is strongly recommended that the user supplied fault string also contain the same text. The user FAULT can get the current statement number from the variable **:_cs:** and should use that instead of hard coding a line number.

The handler for a user FAULT statement must not go to the **:_cs:** line directly, as the :**_cs**: was the FAULT statement and the FAULT statement itself will re-execute, resulting in an infinite loop. Instead, it should go to a program label that will retry the failing operation. Alternatively, if the fault-handler routine can catch up program state to the functionality of the failing user code, the fault handler can go to the next line- that is, go to **:_cs: + 1**.

Here's an example of a user-mode fault handler that does a GOTO back to the line following the fault. Here's the code:

```
{
    window
    output /Starting out, /
    output /current line: :*:_cs:, will FAULT on next line \n/
    input [ nonexistent_file.txt ]
    output /Return from fault service routine at line :*:_cs:\n/
    exit
}
trap /.*/ (:my_trap_string:)
```

```
output /  Now we are in the fault service routine. \n/
output /  Finding the line that we TRAPped on ... \n/
match [:my_trap_string:] /This happened at line ([0-9]+) / (::
:my_line_num:)
output / Found that we trapped on line number :*:my_line_num:.\n/
output /  Going to the line after that \n/
goto / :@: :*:my_line_num: + 1 :/
```

When we run it, we get:

```
# crm trap11.crm

Starting out, current line: 4, will FAULT on next line
  Now we are in the fault service routine.
  Finding the line that we TRAPped on ...
 Found that we trapped on line number 5.
  Going to the line after that
Return from fault service routine at line 6
#
```

thus showing that we can execute fix ups in a TRAP and continue program execution after fixing the problem.

“On to Little Big Horn for glory. We've caught them napping.”

- General George Armstrong Custer, United States Army.

Section 4: Tracing, DEBUGging, and Profiling

CRM114 includes its own command-line debugger. The debugger is unfortunately not currently integrated with GDB or DDD, because of the huge semantic gap between CRM114's overlapped-string data model and most procedural languages. That said, it is still quite possible to debug CRM114 programs.

Tracing, debugging, and profiling are triggered by command-line switches or by the DEBUG statement. On the command line, you can use:

`-d`	immediately drop into the debugger as soon as the first stage of compilation is complete
`-d N`	run N statements, then drop to the debugger
`-t`	standard trace (user-understandable). This includes printing an annotated program listing, so the trace line numbers will match up with the annotated listing line numbers.
`-T`	detailed trace (for systems developers). This includes a very detailed program listing with internal per-line parse information, as well as a gruesomely explicit trace of program execution
`-p`	profile the code; print an execution-time summary on exit

These flags can be combined as desired.

If you know where in your program things are going astray, you can put in a DEBUG statement right in line and your program will automatically drop into the debugger as soon as the DEBUG statement is executed. The DEBUG statement is simple and takes no options:

```
        debug
```

Every time your program executes the DEBUG statement, it will drop into the debugger. If this is in a loop, you will end up hitting “c” (for continue) a lot. A good trick for this is to use a MATCH to decide whether your program's brokenness is imminent; if it is, then drop to the debugger and allow a detailed examination of the state. A short code stanza like this:

```
...
{
      match [:some_var:] /broken_program_recognizing_regex/
      debug
}
```

will save your sanity and speed your debugging. Of course, if your regex is not quite right, you may find yourself debugging your debugging triggers.

A Trace Example

Here's a trace of the “Hello World” program:

```
# cat > hello1.crm
{
        output /Hello, world! \n/
}
# crm hello1.crm -t
Setting usertrace level to 1
Loading program from file hello1.crm
Hash of program: 8F3F31F0, length 34 bytes
Program statements: 6, program length 34

0000 {00} :  --
0001 {00} :  -{-
0002 {01} :    -output-=/Hello, world! \n/=
0003 {00} :  -}-
0004 {00} :  --
0005 {00} :  --
Starting to execute hello1.crm at line 0

Parsing line 0 :
 -->


Executing line 0 :
Statement 0 is non-executable, continuing.
```

```
Parsing line 1 :
 -->  {


Executing line 1 :
Statement 1 is an openbracket. depth now 1.

Parsing line 2 :
 -->  output /Hello, world! \n/


Executing line 2 :
 Executing an OUTPUT statement
  filename >>><<<
  pre-IO seek to >>><<< --> 0
  and maximum length IO of >>><<< --> 8388608
Hello, world!

Parsing line 3 :
 -->  }


Executing line 3 :
Statement 3 is a closebracket. depth now 0.

Parsing line 4 :
 -->


Executing line 4 :
Statement 4 is non-executable, continuing.
Finished the program hello1.crm.
#
```

Notice that even at this level, the trace is rather detailed. The one good thing about this trace is that if you need to know something about your program, you'll probably find it in the trace.

A Profiling Execution Time Example

CRM114 can profile the execution time of your program. Because of the granularity of most system timers (for an x86 running Linux, typically 10 milliseconds per tick), this gives only a rough estimate of the time spent in each statement, but it's still very handy for program optimization.

For our example, we'll use our "copy the dictionary" program.:

```
{
        window
        input [ /usr/share/dict/linux.words ]
        output [ cloned.words ] /:*:_dw:/
}
```

We'll run it with profiling turned on; that's a **-p** flag on the command line. Profiling output is "smart" – only statements with a nonzero execution time will be listed at all. For our example, we'll also get a bash **time** measurement, to compare overall timing granularity.

```
# time crm clonedict.crm -p

         Execution Profile Results

  Memory usage at completion:    4992010 window,       3242 isolated

  Statement Execution Time Profiling (0 times suppressed)
  line:      usertime   systemtime    totaltime
     3:           2            6            8
     4:          12            8           20

real    0m0.496s
user    0m0.147s
sys     0m0.153s
#
```

Note that line 3 (the input) took a relative time of 8 units, while line 4 (output) took 20 time units.

The overall "usertime" of our statements was 14 units, corresponding reasonably with **time**'s 0.147 seconds of user-mode CPU time; the "systemtime" on our statements was also 14 units, corresponding moderately well with **time**'s measurement of 0.153 seconds. This also reveals that our system clock update frequency is 100 Hz.

Remember, profiled times are only approximate; use them as an aid to program optimization, not as gospel. Another useful item in the profile is the final memory usage, for the data window and for isolated variables. This will give you a good idea of how much memory your program actually needs.

A Debugging Example

The built-in terminal-oriented CRM114 debugger is a model of simplicity (or, if you prefer, paucity). The commands can be listed right from the debugger itself:

```
# crm -d '-{ }'
CRM114 Debugger - type "h" for help.  User trace turned on.

crm-dbg> h
a :var: /value/ - alter :var: to /value/
b <n> - toggle breakpoint on line <n>
b <label> - toggle breakpoint on <label>
c - continue execution till breakpoint or end
c <n> - execute <number> more statements
e - expand an expression
f - fail forward to block-closing '}'
j <n> - jump to statement <number>
j <label> - jump to statement <label>
l - liaf backward to block-opening '{'
n - execute next statement (same as 'c 1')
q - quit the program and exit
t - toggle user-level tracing
T - toggle system-level tracing
v <n> - view source code statement <n>

crm-dbg>
```

which gives you the basic operations needed to debug a program during execution. For our example, we'll debug the “substitute texts” program. This program replaces all instances of its first argument with its second argument. Here's the code:

```
{
      match (:the_match:) /:*:_arg2:/
      alter (:the_match:) /:*:_arg3:/
      liaf
}
accept
```

and an example run:

```
#  crm debug_1.crm vanilla chocolate
I would like to order a hamburger,
with a vanilla shake, a side of fries,
and a double expresso with a shot of vanilla syrup.
```

```
return, then ctrl-D
I would like to order a hamburger,
with a chocolate shake, a side of fries,
and a double expresso with a shot of chocolate syrup.

#
```

which works as expected. But- this program can fail – consider this usage (yes, I misspelled "necromancy" intentionally. Please forgive me):

```
# crm debug_1.crm romance necromancey
Perky Goth: We put the romance back in necromancey
return, then ctrl-D
```

and there the program hangs, or at least seems to- stuck in an infinite loop. (Actually, it will eventually error out, but that can take a long time.) Let's use the debugger to see what's going on; we'll run 30 statements, then drop to debug (done with a **-d 30** on the command line), then we'll see what the default data window looks like (with **e**valuate **:*:_dw:**) :

```
# crm debug_1.crm romance necromancey -d 30
Perky Goth: We put the romance back in necromancey
CRM114 Debugger - type "h" for help.  User trace turned on.

crm-dbg> e :*:_dw:
expanding:
 Perky Goth: We put the necnecnecnecnecnecnecromanceyyyyyyy back in
necromancey


crm-dbg>
```

So, there's our problem: our replacement pattern evaluates to contain itself, so we recursed our way into a hole that will only exit when we run out of space in the default data window and cause a FAULT. Now that we know what we did wrong, the fix is obvious – we just need to prevent repeated expansion of the same text. We can do that with any match control flag that forces forward movement in the match, like **<fromnext>**, **<fromend>**, or **<newend>.** Our fixed program is:

```
{
      match (:the_match:) /:*:_arg2:/ <fromend>
      alter (:the_match:) /:*:_arg3:/
      liaf
}
```

```
      accept
```

and when we run it, we get:

```
      # crm debug_2.crm romance necromancey
      Perky Goth: We put the romance back in necromancey
      return, then ctrl-D
      Perky Goth: We put the necromancey back in necnecromanceyy
      #
```

exactly what we thought we wanted.

Different people have different debugging styles. Some people use a debugger “passively”, setting breakpoints and watching variables, but not interfering with the program's execution. Others change control variables, jump around in the code, and otherwise take a very active part in program execution. You should use whatever debugging style suits you and that you find effective.

“In Japan, employees occasionally work themselves to death. It's called *Karoshi*. I don't want that to happen to anybody in my department. The trick is to take a break as soon as you see a bright light and hear dead relatives beckon.”

- Scott Raymond Adams

Section 5:
Idioms and Tricks

In this section we'll discuss some CRM114 idioms and tricks. Some of these tricks implement subtle but useful control effects, and others are speed improvements.

Classifying Languages Without Spaces (Asian Languages)

Many Asian languages (Japanese and Chinese, and probably others) have a printed representation that does not use a space character to delimit words, and sometimes (not always!) use a newline to separate lines of text. This means that the default **`[[:graph:]]+`** regex in LEARN and CLASSIFY will take each line of text and use that as a single "word", then combine those words into "features". Unfortunately, this means that unless your Asian-language spam uses the exact same words over and over, with exactly the same line breaks and pagination, you won't be able to do any matching on them.

What makes matters even worse is that there isn't a single standard way to encode non-Western fonts; Japanese has four accepted "standards" for character representation in email alone. There are several ways to deal with these problems.

Altering the Tokenizing Regex

The simplest way (which may be accurate enough) is to detect somehow that the email is in a no-spaces language, and if that occurs, to use a different feature regex for LEARN and CLASSIFY; one that gets 'enough' information out that the features are meaningful. Typically, this means a word should be 1 to 3 bytes long (feel free to experiment; tests using OSB on English (not Asian) languages show a very acceptable classification success rate based solely on character strings of a few letters – 97% for strictly letter pair "words" such as the tokenizing regex **`/../`** . This is the equivalent of wchars but allowing any character (including blanks and newlines) in any position. Using a slightly smarter regex like **`/[[:graph:]][[:graph:]]/`** gives 98% accuracy).

Using a Pretokenizer

Another, better method is to use a pretokenizer that inserts spaces into spaceless text at "reasonable" places. Note that the "reasonable" places don't need to be perfect; 95% accuracy is probably just fine, as long as it's repeatable. A pretokenizer like **`kakasi`** (free and GPLed open source) is a good bet for Japanese and possibly for Chinese. Fortunately, CRM114 is 8-bit-safe, so as long as the pretokenizer gives repeatable results (that is, for similar input strings the output is also similar), the resulting "words" will be featurizeable and should classify without a problem even if the pretokenizer doesn't transliterate to the Latin-1 character set as **`kakasi`** does..

In particular, using **`kakazi`** with the following flags seems to give reasonably-tokenized ASCII that CRM114 can use for LEARN and CLASSIFY with the default **`[[:graph:]]+`** tokenizer regex:

```
      kakasi -Ha -Ka -Ja -Ea -ka -s < asian_text_file.txt
```

You can get kakasi from:

```
      http://kakasi.namazu.org/
```

and you may also find it useful to read:

```
      http://www.math.kyoto-u.ac.jp/~nakajima/HowTo.html
```

for more hints on dealing with non-Western texts (in particular, Japanese). This page is more about displaying text than spam filtering, but it's useful nonetheless.

Create Your Own Pretokenizer

Lastly, you could write a pretokenizer yourself. Efficiently tokenizing a language needs a deep understanding of that language, so it's not a task to be undertaken lightly. On the other hand, if you don't know that particular language, then anything you get in those langauges is likely to be spam, and hence discardable.

If you do succeed in creating a pretokenizer for a language that doesn't tokenize easily with a regex, please do consider putting it up as GPLed software. The world may end up thanking you.

WINDOW Is Too Slow For Me

When WINDOWing, the default setup is optimized for correctness in all possible cases, not for raw speed. In particular, there are two situations that often crop up when using WINDOW that can be optimized by the programmer by using domain-specific knowledge. Specifically, when the source of characters for the WINDOWing is known not to not be accessed by any other code, it's possible to speed up WINDOWing by a huge amount.

The two biggest speedups are when either:

- The program WINDOWs from `stdin`, and only WINDOW statements will access `stdin,` so reading all of `stdin` to EOF will not keep a later program from it's rightful inputs.
- The program WINDOWs from an ISOLATEd variable, and no other MATCH-type accesses to the source variable's characters are made in the program.

Both of these situations allow alternative implementations that can run much more quickly than the default implementation.

WINDOWing from stdin

By default, WINDOW reads a single character at a time from `stdin,` then runs the add regex against the single character. If the add regex didn't match, another character is read, and the add regex is run again. This process repeats until the add regex is satisfied. This gives correct behavior when other INPUT statements or even other programs are sharing `stdin`. Unfortunately, this is slow (character-at-a-time input is inefficient and executing a regex repeatedly for each character is inefficient). This slow path is necessary so that WINDOW doesn't read too many characters, especially characters meant to be read by subsequent programs.

If only WINDOW statements will access `stdin` in any way (including "access" after the CRM114 program exits) then it is possible to have CRM114 read in much larger chunks of `stdin`, and then run the add regex only once. This can speed up execution considerably; probably enough to make it worth your while to arrange the calling program such that input can always be read in a large chunk, preferably to EOF.

To tell the WINDOW that it's acceptable to read in more than one character at a time from `stdin`, use the **`<bychunk>`** or **`<byeof>`** flags. The first, **`<bychunk>`**, says "read a conveniently large sized block of text", the second **`<byeof>`** means "read all the way to EOF". In both cases, a hidden internal buffer is used to save the unused characters. Because there's no way to "unread" more than a single character, this means that possibly <u>all</u> of the data waiting in `stdin` may be read and become inaccessable to anything except WINDOW statements in the current program.

Here's an example – we'll WINDOW through <u>Hound of the Baskervilles</u> (all 320K of it) one character at a time (the default), and then **`<bychunk>`**. In both cases, the add regex is the word “Huguenots”, which appears only in the final paragraph of Hound.

```
window
{
      window /.*/   /Huguenots/
      output / found it\n/
}
```

In this case, the default WINDOW code ran for about an hour and didn't finish:

```
# time crm houndsearch1.crm < Hound_of_the_Baskervilles.txt
go have , dinner, relax, then hit  ctrl-C here

real    54m22.849s
user    0m0.011s
sys     0m0.013s
#
```

Note that this code ran for almost an hour, and never finished; we hit ctrl-C instead. Clearly this isn't acceptable performance.

To see a contrast, we add **`<bychunk>`** to the code:

```
window
{
      window /.*/   /Huguenots/ <bychunk>
      output / found it\n/
}
```

and get a much faster result:

```
# time crm houndsearch2.crm < Hound_of_the_Baskervilles.txt
 found it
real    0m0.177s
user    0m0.114s
sys     0m0.015s
#
```

which is something well over 30,000 : 1 speedup – and might be a lot more, as the default character-at-a-time mode didn't even complete.

WINDOWing from an Internal Variable

In the case of WINDOWing from an internal variable, one can “deconstruct” WINDOW to its base components, often with a major speedup, if the algorithm being implemented can be restructured so that the following are all true:

1. The sourcing variable must not be accessed in any way except by the deconstructed WINDOW operator.
2. The algorithm does not care what remains in the sourcing variable when the WINDOW completes (though this can be relaxed at significant speed loss)
3. The algorithm must not depend on the flags **`<eofretry>`** or **`<eofmatches>`** , although these can also be relaxed with hand-carved code and some speed loss.

If all of these are acceptable, then the deconstruction of WINDOW for a sourcing variable is exactly in parallel to the **`<bychunk>`** or **`<byeof>`** modes above. Here's skeleton code where **`:win:`** is the windowed variable, **`:src:`** is the source variable, and the vars **`:cut_regex:`** and **`:add_regex:`** are the deletion and addition marker regexes.

```
{
      #   start with deleting the deletion regex stuff
      match (:deletion_zone:) [:win:] /.?*:*: cut_regex:/
      alter (:deletion_zone:)  //
      #    now, find the stuff we want to add
      match (:addition_zone:) [:src:] /.?*:*:add_regex:/
      #      and add it to the end of the working window.
      match ( :: :addition_point:) [:win:] /.*($)/
      alter (:addition_point:) /:*:addition_zone:/
      alter (:addition_zone:) //
    [...     and the rest of your code goes here .....]

}
```

The downside of this deconstructed WINDOW code is that it doesn't allow easy ways to add the flags **`<eofretry>`** or **`<eofmatches>`**, nor does it make it easy to tell what the reason for a FAIL was. However, it may run considerably faster than the “cover all cases” WINDOW code will run.

WINDOWing with a “Cover” Variable

If neither of the above speedup methods will work for you, a simple change to your program's code can sometimes get you most of the above speedups (but does not release memory back into the common use pool – if your program is chronically a memory pig, this won't help at all).

The reason this speedup works is that repeated on-the-fly reclaims of single characters is not very efficient; if we don't need to reclaim the space, we can force the reclaimer to not run by MATCHing a second variable right on top of the same memory as the to-be-windowed string, and then simply ignoring that variable (well, we ignore it- but the reclaimer doesn't!)

The code for creating a “cover” variable is very simple; just MATCH another variable onto the WINDOWed variable:

```
# create a “cover” variable on top of my real variable
match (:my_cover:) /.*/ [:windowed_variable:]
{
      window [:windowed_variable:]
      .....
}
#    and release the cover if we want to reclaim the memory
isolate (:my_cover:) //
```

This technique is easy to try, and easy to back out if it turns out your program needs every last bit of memory

Finding lines that “Don't Contain”

An interesting problem arises when we want to efficiently find lines of text that don't contain a certain pattern. Of course, one way to do it is to WINDOW through the text, but there are other ways.

For example, consider the `du` utility. Running `du` produces a list of the size of all directories, subdirectories, etc. starting in the current directory. Running `du` on a disk's mount point is a good way to find out where all of that expensive disk space went.

The problem with `du` output is that it's excruciatingly detailed; it gives every directory its own line of output, and so the useful information (say, just the grand totals in the current directory) is buried deeply in the `du` output. What we want is a way to see only the top-level directory information – that is, to see only the lines that contain <u>only one</u> slash, and there's no way to get that directly from `du`. Here's a sample of `du` output:

```
...
24      ./etc/log.d/scripts/logfiles/xferlog
24      ./etc/log.d/scripts/logfiles/samba
16      ./etc/log.d/scripts/logfiles/autorpm
24      ./etc/log.d/scripts/logfiles/up2date
112     ./etc/log.d/scripts/logfiles
```

```
        ...
```

Note that you cannot effectively see the forest for the trees. Note also that `grep` can't easily do this job. What we want is just to see the top level directories, and `du` doesn't have a flag to only show that. We need a program to filter the output of `du` to only show top level directories - that is, lines that contain only a single slash, but not more than one slash.

Here's a program that looks like it should work, but doesn't....

```
        {
                #  THIS CODE DOES NOT WORK AS EXPECTED !!!!!    DO NOT USE!!!
                match (:l:) <absent nomultiline fromend> /^.*\/.*\/.*$/
                output /:*:l:\n/
                liaf
        }
```

The problem here is that **`<absent nomultiline>`** doesn't mean “find the first case where this isn't true”. Rather, it means “test the input globally, and succeed only if no line is found that contains at least a pair of slashes.”

What we really mean is to find lines containing just one slash; that is, a line that contains
1. the beginning of the line,
2. then zero or more non slash characters,
3. then a single slash,
4. then zero or more non-slash characters,
5. then the end of line.

As an aid to understanding, sometimes it helps to write out a little cheat-table when creating a complex regex like this one. This cheat table has just two columns – what we want, and how we get it:

What We Want	***How We Get It***
Beginning of line	**`^`**
Then zero or more nonslash characters	**`[^\/]*`**
Then a single slash	**`\/`**
Then zero or more nonslash characters	**`[^\/]*`**
Then end of line	**`$`**

Expressed this way, we can easily write the corresponding regex just by reading down the “how we get it” column in the table. The result reads like modem noise but is correct:

```
      ^[^\/]*\/[^\/]*$
```

The corrected program is:

```
      {
            match (:l:) <nomultiline fromend> /^[^\/]*\/[^\/]*$/
            output /:*:l:\n/
            liaf
      }
```

which gives the desired result (excerpted below).

```
      ...
      2640    ./tre-0.7.0
      4192    ./crm114-20040409-BlameMarys.src
      51572   ./ctst
      2760    ./tre-0.7.2
      ...
```

Checksumming Program Source Code

Sometimes, you need a quick integrity check of a program or data file, and you'd rather not depend on having a strong checker like MD5 or SHA-2[31] installed. To do this, use the HASH statement. Note that this isn't a cryptographically secure check (it most certainly is not cryptographically secure). All this does is to assure your (or your programs) that nothing stupid-silly has damaged a program file.

In the case of the currently executing program, this is done for you automatically by CRM114 during the compilation phase- the internal variable **`:_pgm_hash:`** contains the hash of the current program after all INSERTs are completed.

So, for a quick integrity check, when your program is invoked in "version" mode or "help" mode, you may find it useful to also print out **`:*:_pgm_hash:`** as a simple check to see if your program has had any changes made to it. This is great for catching liars who claim "We didn't touch your program, really we didn't".

31 SHA-1's been broken, so it's no longer reasonable to consider it "strong". Whether it's "strong enough" for your purposes is another matter altogether.

How Much Memory is My Program Really Using?

Some operating systems place rather stringent limits on program storage space; occasionally the OS will not even allow a CRM114 program to run up to the internal anti-DoS limits[32]. In these cases, you can diagnose the problem by using the **`:#`** evaluator on the two important data strings - **`:_dw:`** (the data window) and **`:_iso:`** (the isolated data area).

Just throw in the occasional OUTPUT statement that gets the string length of these two data areas and you will see how memory usage moves around during program execution (you can also evaluate these string lengths from inside the debugger).

Here's an example to show how to find out how much memory is really in use:

```
{
      isolate (:my_mem_status:) //
      eval (:my_mem_status:) \
            / DW used= :#:_dw: , Isolated used= :#:_iso:/
      output /:*:my_mem_status: \n/
}
```

When we run it, we get:

```
# crm memuse.crm
One Two Three
return, then ctrl-D
 DW used= 14 , Isolated used= 2526
#
```

What's using all that isolated space? Mostly, it's the environment variables that get loaded in from the shell. Let's use **`-e`** to not load environment variables. Now we get:

```
# crm memuse.crm
One Two Three
return, then ctrl-D
 DW used= 14 , Isolated used= 383
#
```

showing that CRM114 needed 383 bytes of isolated data space just to store its own bookkeeping variables in a programmatically-accessible area, plus the variable space needed for one string containing the current memory usage.

32 For example, some BSD variants are often configured with very tight per-process limits.

Demonizing a program for even more speed

Sometimes you're in a programming situation where throughput is absolutely critical. If that's the case for you, then there's a programming trick in CRM114 that can help you. It's called *demonizing* the program, so that the program is launched once, and then it executes itself on multiple input files without doing a full reinitialization, re-reading the program source, re-preprocessing the source, re-JITting the statements, etc. This can speed up program execution time for short programs by an order of magnitude.

There are at least two methods of demonization possible – looping and spawning.

Looping Demonization

In the looping form of demonization, the program starts up and runs one batch of data (say, one email message), and then (via either a pipe or stdin) is given another batch of data. It runs that batch of data, and then is given yet another batch of data.

This is the most efficient method in CRM114- but it has a security flaw, in that previous chunks of data are not guaranteed to be isolated from subsequent chunks. Thus, there is a possibility of:

- Annoying bugs, when a seldom-used option is not reset properly so that a number of subsequent batches of data are also processed with the seldom-used option.
- Data leakage, where a bug in the program lets data hang around from one chunk of data to the next.
- Fiddly little memory leaks. CRM114's internal libraries are well tested to not leak memory, but there's always the possibility that something has been missed. This might mean that a looping demon may eventually run out of some necessary resource.

On the good side, a looping takes complete advantage of everything that the JIT compiler and memory-map cacheing system can do. Because the statements are fully executed, the JIT compilation runs to completion and the code is as quick as a JIT can make it. Looping demons also are easy to write – the usual scheme is to use **`mkfifo`** create a named fifo in **`/tmp/`** that is fed the names of the files containing batches of data to run (such as the filenames of email messages that need filtering) and have the looping demon read that fifo.

A good example of this is seen in the section on asynchronous processes and SYSCALL, on page 133.

Spawning Demonization

In the spawning form of demonization, the main program runs a very simple loop that waits for a new batch of data to become available, and then SYSCALLs the actual deal-with-the-data code (creating a forked minion process). Because the process that actually touches user data is used only once and then thrown away, the chance of having problems with option-reset bugs, data leakage, or memory leaks is greatly diminished.

However, unless the program is coded to execute each statement in the SYSCALLed minion path before the first actual SYSCALL (thereby forcing the main program to have completed JIT on every statement) the JIT compiler will end up recompiling the statements in the minion's code path again and again for each minion's execution. This decreases speed of the spawning demonization slightly. Spawning demons also don't get to share their memory-mapped caches, so unless the main program does at least one CLASSIFY on the .css files to be used (thereby causing the cacheing of the .css data) each .css file will be remapped again and again, rather than mmapping once and cacheing thereafter as a looping demon will.

“A good example is the best sermon”

- Benjamin Franklin

Section 6: An Email Filter Example

In this section we'll actually construct a simple antispam email filter. This is a fully functional anti-spam filter, and it will show how all of the pieces of CRM114 fit together to make it easy to create and customize whatever filtering structures you need. It's this ability to create complex filters easily that make CRM114 valuable.

We'll assume that the system is providing us with `procmail` to use as an interface to the host's MTA (Mail Transfer Agent), because `procmail` is in common use and already can do things like properly lock mail spools and access mail folders. We'll also assume that our MUA (Mail User Agent – the program that the human uses to actually read mail) is capable of sorting email based on keywords in the header or of using mail folders. We'll sort mail into two types – GOOD and SPAM – and tell our calling program what type each message is (via error codes) and also add a "tag" to the subject of each spam message..

Overall Design of this Example

Let's assume we want a multi-level filter. First, we will let CRM114 read `stdin` to get the incoming mail message. Then, the program should check if the email contains any item on our special word list (words or phrases that we've defined to be either good, or bad; as soon as one of these is encountered, we know whether the email is spam or good). If the special word list does not have any matches, the filter should use a CLASSIFY statement to make a best guess. Because the statistical CLASSIFY system needs to be trained with examples of good email and spam email (preferably trained only when it makes a mistake, known as TOE (Train Only Errors) training, we will also need to have a side utility that lets us train good email, and another that lets us train spam.

Rather than presenting one big program, we'll go through the blocks of code in manageable pieces, as subroutines. The pieces will be a the special-word-list handler, the statistical CLASSIFYer, the "disposal methods" for good email and spam email, and lastly a main program to tie it all together. As this is a very simple program, we'll use subroutines without argument transfer; all routines will need to agree on the important variable names.

In our case, there are just two important global variables: the **:_dw:** which holds the incoming text, and a variable named **:conclusion:** which will take three possible values:

unknown	We don't know yet if this is good or spam (initial value)
good	We've decided this is good
spam	We've decided this is spam

There will also be three auxiliary files:

good.css	the statistics file for examples of good email
spam.css	the statistics file for examples of spam email
trigger_words.txt	our trigger words. Each word is prefixed with a plus **+** (for good) or a minus **–** (for spam) .

Finally, we need to define how our mail filter returns its output so the mail reading system can sort the good mail from the spam. Since we probably want to have as few changes to good mail as possible (and because this is just a little example program) we'll limit ourselves to just ONE in-text message, and (for procmail users) also use a return code. For the in-text message, we'll prefix the SUBJECT text header in the mail with the string [[SPAM]] if the message is believed to be spam, and leave it unchanged for good email. We'll also return a 0 exit code for good email, and a 1 for spam, and 99 for "system error".

Example Whitelist and Blacklist Code

All of our whitelisting and blacklisting is done in one routine -- `:triggerword_check:` This routine reads in the `trigger_words.txt` file, then uses a MATCH/LIAF loop to step through each line. The first character (which is always either **+** or **-**) is split in the MATCH from the trigger word on rest of the line, and checked to see if the input text (in **:_dw:**) contains the trigger word. If it doesn't contain the current trigger word, we get the next trigger word. If it does, then we set the `:conclusion:` var appropriately and return from the routine.

The local variable **:trigwords:** will contain the entire set of trigger words read as a single block from `trigger_words.txt.` The variable **:trigword:** will be one word, and we then set the variable **:goodbad:** to either **+** or **–** (for "good" or "spam").

Because the trigger word strings will both be matched by a regex (to extract the **+** or **-**) and then used as a regex pattern themselves, we need to be a bit careful, as characters like **.** (a period) will retain their regex-specialness capabilities; for example, we need to use \Q and \E on "mit.edu" to force the period to match a literal period.

On the other hand, we can also use CRM114's TRE approximate regex matching capability to match all sorts of misspellings of a spammish word like "Viagra", remembering to wrap the entire text we want approximate matching on in parentheses. We can even use the sequential nature of this algorithm (checking the trigger words in a specific sequence) to *accept* email that contains "viagra" (spelled correctly), but reject email that contains "camouflaged" or misspelled Viagra (in this case, with up to two spelling errors).

Here's the code for the triggerword_check routine:

```
:triggerword_check:
{
      input (:trigwords:) [triggerwords.txt]
      {
            #           Get the next trigger word (there's one per line)
            match [:trigwords:] (:: :goodbad: :trig:) \
                  <nomultiline fromend> \
                   /(\+|-)(.*)/
            #
            #     Is that trigger word in the input text?
            {
                  match <nocase> /:*:trig:/
                  #              yes, :trig: is in the unknown text.
                  {
                        match [:goodbad:] /-/
                        #           it's bad
                        isolate (:conclusion:) /spam/
                        return
                  }
                  isolate (:conclusion:) /good/
                  return
            }
            #  no, this trigger word did not match.  Go try again.
            liaf
      }
      #  The "next triggerword" match finally failed.  Inconclusive...
      isolate (:conclusion:) /unknown/
      return
}
```

For a test, we can wrap this program in a test harness – this simple main routine does the default read-to-EOF, calls **`:triggerword_check:`**, and prints out the value of the variable **`:conclusion:`** so we can tell if the code is working or not.

```
      #!/usr/bin/crm
      {
            call /:triggerword_check:/
            output /Result is ":*:conclusion:" \n/
      }
```

Here's the text of the `triggerwords.txt` file. Note that we are using the "approximate regex after exact regex" trick to allow the exact word "viagra" but not mutated versions of the word (hence **`+viagra`** followed by **`-(viagra){~2}`** on the next line):

```
      +CRM114
      +viagra
      -(viagra){~2}
      +\Qmit.edu\E
      -casino
```

Now, we'll test this first part of the filter:

```
      # crm minifilter1.crm
      You can write your own customized filters in the CRM114 language.
      It's very powerful
      Result is "good"
      # crm minifilter1.crm
      For example, the word viagra is not by itself a sign of spam
      Result is "good"
      # crm minifilter1.crm
      But people who camoflage it as vi@gra are usually spammers
      Result is "spam"
      # crm minifilter1.crm
      Note that casino is a spammish word
      Result is "spam"
      # crm minifilter1.crm
      But casino is later on the triggerwords list, and mit.edu occurs
      before casino, so this isn't spam
      Result is "good"
      # crm minifilter1.crm
      If we avoid all the trigger words, the result should be "unknown".
      Result is "unknown"
      #
```

We now have the first level antispam filter coded, documented, and tested.

Example CLASSIFY and LEARN Code

The next routine in our program is `:statistical_check:` . This routine uses two statistics files, `spam.css` and `good.css`, to get a best guess of whether the unknown text is good or spam when the text doesn't contain any trigger words.

To keep the match set small, but maintain good accuracy, we'll use the OSB classifier, with the **<unique>** and **<microgroom>** options. Unique means "each feature found is counted only once, even if it appears many times in the input text", and microgrooming means to automatically discard old, low-relevance features when the amount of remaining storage in the .css file drops dangerously low. We will use the default tokenizer of whitespace-delimited words, as that works reasonably well for most English text.

As all routines in this program agree that the unknown text is in the data window **:_dw:** and the spam versus good result is in **:conclusion:**, we don't need to pass any parameters in or out.

Here's the code the **:classify_check:** routine:

```
:classify_check:
{
#      nest down one level to catch the fail...
      {
            classify <osb unique microgroom> ( good.css | spam.css )
            isolate (:conclusion:) /good/
            return
      }
      isolate (:conclusion:) /spam/
      return
}
```

If we then test this code using the same test harness main program that we used for the previously tested `:triggerword_check:` routine, we find that everything is good! The second thing we find is that this program, although correct, generates runtime warnings that it cannot find the files `good.css` and `spam.css`.

The first problem is that with no examples in the .css statistics files, all of the result probabilities are exactly 0.5 and the resulting pR's zero. Because CRM114 obeys "when in doubt, nothing out", this means all email will fail the CLASSIFY and be marked as spam. We need to supply some example text if we want to get reasonable results from the classifier. The second problem is that our .css files don't exist yet; we need to create them. Both of these problems will be solved by actually putting some example text into the .css files.

To put our example texts into the .css files, we need two helper utility programs- one for example spam emails, and one for example good emails. Both programs are just one statement long, so we can use the command line to enter the program:

```
      crm '-{ learn <osb unique microgroom> (good.css) }'
```
and
```
      crm '-{ learn <osb unique microgroom> (spam.css) }'
```

(note to the reader: we could put these programs into files just as easily.)

Using these command line programs to learn some sample text (cut from the text sections of actual good mail and spam the author received in the previous 24 hours), and then checking for overlaps with `cssutil`, we get:

```
# crm '-{ learn <osb unique microgroom> (good.css) }' < samplegood.txt
# crm '-{ learn <osb unique microgroom> (spam.css) }' < samplespam.txt
# cssdiff good.css spam.css
Sparse spectra file good.css has 1048577 bins total
Sparse spectra file spam.css has 1048577 bins total

 File 1 total features            :         6399
 File 2 total features            :         6386

 Similarities between files       :          148
 Differences between files        :         6244

 File 1 dominates file 2          :         6251
 File 2 dominates file 1          :         6238
#
```

You may notice the close balance between the feature counts in these two files. This was genuinely a matter of luck (and you'll just have to take it on faith because I can't show you the email; some of it was confidential in nature) .

Now for the moment of truth. We'll feed our filter the opening text of this chapter and see what the filter thinks of it:

```
      # crm minifilter2.crm
      In this section we'll actually construct a simple antispam email
      filter.  It won't be a complex filter, but it will be functional,
      and it will show how all of the pieces of CRM114 fit together to
      make it easy to create and customize whatever filtering structures
      you need.  It's this ability to create complex filters easily that
      make CRM114 valuable.

      We'll assume that the system is providing us with procmail to use as
      an interface to the MTA (Mail Transfer Agent), because procmail
      already can do things like properly lock mail spools and access mail
      folders.  We'll also assume that our MUA (Mail User Agent – the
      program that the human uses to actually read mail) is capable of
      using mail folders; we'll simply sort mail into two folders — GOOD
      and SPAM.
      Result is “good”
      #
```

Well, that's comforting. Let's hit this filter with the text of a spam (with identifying information removed only in the version printed; the filter itself got un-redacted text):]

```
# crm minifilter2.crm
Your Special * ******* Introductory Offer

Get 6 bottles free with the purchase of 6 bottles!
That is 12 Bottles of Premium Wine
Only $4.99 per Bottle
(normal retail average of $16/bottle)

Click Below to Order
http://*********.com/redir.php?id=***&e=***@****.com
If the above link is not active, please cut and paste the
entire address into your browser

With your first order you will receive
our ********* ******** Tabletop Wine Opener...
a $139.95 value -- Absolutely F r e e!
Result is “spam”
#
```

which is also correct; this is spam. Interestingly, the spam learning examples did not contain any spam trying to sell wine, nor did the good mail examples contain any reference to spam filtering. That's the nice thing about statistical filtering based on the Bayesian or Markovian models; exact matches are not necessary. Sometimes it's a little spooky how well it works.

Example Dispatching on the Routine Results

We now have a set of routines that will give us a result, either “good” or “spam” in the pre-agreed variable **:conclusion:**. We now need to check this value, and if it's “good”, print out the input text, and exit with an exit code of 0. If it's “spam”, we need to alter Subject: header to contain **[[SPAM]]**, print out the text, then exit with an exit code of 1. If the mail was malformed and didn't include a Subject header, we'll add one. For now, we'll disregard the various perversions of how headers can be mangled by sufficiently aggressive spammers. Finally, if **:conclusion:** isn't one of “good” or “spam”, just return, so the main program can try a different way of deciding if the text is good or not.

Here's the code:

```
:dispatch_result:
{
      match [:conclusion:] /good/
      accept
      exit /0/
}
{
      match [:conclusion:] /spam/
      #     it's spam... find the subject header and change it.
      {
            match (:subj:) <nocase nomultiline> /^Subject:/
            alter (:subj:) /Subject: [[SPAM]]/
            accept
            exit /1/
      }
      #     what if there WAS no subject header?
      {
            alter (:_dw:) /Subject: [[SPAM]]\n:*:_dw:/
            accept
            exit /1/
      }
}
#     it's neither- just return
return
```

To test it, we'll need a slightly different test harness- this one uses **`:_arg2:`** (the first user-settable command line argument) as either "good" or "spam":

```
#!/usr/bin/crm114
{
      isolate (:conclusion:) /:*:_arg2:/
      call /:dispatch_result:/
}
```

(Note: we could have written this as a one-line routine using a command-line value for the **`:conclusion:`** variable; we'll leave that as a traditional "exercise to the reader")

Let's run it with a few examples – spam, good, and "neither" (to show that if we don't have a decision, it does no harm to call this routine):

```
# crm minifilter3.crm good
From: Smith
To: Jones
Subject: Want coffee?

Want some coffee?
hit return, then ctrl-D
From: Smith
To: Jones
Subject: Want coffee?

Want some coffee?
# echo $?
0
# crm minifilter3.crm spam
From: Jones
To: Smith
Subject: Want to be better?

Click here!
hit return, then ctrl-D
From: Jones
To: Smith
Subject: [[SPAM]] Want to be better?

Click here!
# echo $?
1
# crm minifilter3.crm neither
foo bar baz
# echo $?
0
#
```

That all looks correct. Good mail passes through unchanged, while spam gets the subject tagging as [[SPAM]] and also gets an exit code of 1, and nothing at all happens if the mail wasn't yet known. This is particularly easy to integrate into mail processors like `procmail` as well as straightedge mail readers that sort mail based on the subject line.

Example Main Program

Now it's time to write our main program. Because we're using CRM114's automatic read-to-EOF to get the input text, we don't need to visibly do any I/O in the main program. We just initialize things, call `:triggerword_check:`, call `:dispatch_result:`, then (assuming `:dispatch_result:` didn't exit the whole program) call `:classify_check:`

and `:dispatch_result:` again.

```
#!/usr/bin/crm114
{
      isolate (:conclusion:) /unknown/
      call /:triggerword_check:/
      call /:dispatch_result:/
      call /:classify_check:/
      call /:dispatch_result:/
      output / ERROR – this program should never get to here! \n/
      exit /99/
}
```

[[The other three routines go here... but you've seen them already. The full text of minifilter.crm is in the Appendices]]

Notice how pleasantly short the main program is. This is a virtue; it's easy to see exactly how this filter works. There's no black magic, and it will be easy to test and modify.

Testing Our Example Spam Filter

Let's test our example spam filter. This is a reasonable email that I might well send out around lunchtime:

```
# crm minifilter.crm
From: Bill
To: Paul
Subject: Lunch

I'm hungry.  I didn't bring lunch.  Want to go out?
I'd be psyched for chinese, but easy to outvote.

Should we get Darren and Adam as well?

I can go any time.
        – Bill
hit return, then ctrl-D
From: Bill
To: Paul
Subject: Lunch

I'm hungry.  I didn't bring lunch.  Want to go out?
I'd be psyched for chinese, but easy to outvote.

Should we get Darren and Adam as well?
```

```
I can go any time.
        - Bill
# echo $?
0
#
```

Perfectly reasonable – it's not spam, it didn't get tagged as spam, and the exit code is a 0.

Let's try again with some genuine spam – a rather difficult test, actually, considering that the spammish payload is quite short and the camouflaging text quite long. This bit was fresh today (as of this writing), and redacted in the version you see here to protect the guilty:

```
# crm minifilter.crm
From: "Dishonoring P. Glinting" <andranik@***************.com>
To: *** <***@***.com>
Subject: re: disbud about

Only an artist can interpret the meaning of life.
A line will take us hours maybe Yet if it does not seem a moment's
thought, our stitching and unstinting has been naught.
Do not mind anything that anyone tells you about anyone else. Judge
everyone and everything for yourself.
See details
http://a116.noprescrptionsecure.com
If you don't learn to laugh at trouble, you won't have anything to
laugh at when you grow old.
hit return, then ctrl-D

From: "Dishonoring P. Glinting" <andranik@***************.com>
To: Wsy <wsy@merl.com>
Subject: [[SPAM]] re: disbud about

Only an artist can interpret the meaning of life.
A line will take us hours maybe Yet if it does not seem a moment's
thought, our stitching and unstinting has been naught.
Do not mind anything that anyone tells you about anyone else. Judge
everyone and everything for yourself.
See details
http://a116.noprescrptionsecure.com
If you don't learn to laugh at trouble, you won't have anything to
laugh at when you grow old.

# echo $?
1
```

```
        #
```

The spam has been correctly tagged (in the Subject: line) , and the exit code has been set correctly as well.

Teaching the Example Filter

A simple filter such as this will make a lot of mistakes. The advantage of such a filter is that you can easily modify it to suit your needs (putting your lawyer's name into the whitelist, for example) as well as using the single-statement “helper” programs to LEARN any mis-classified spam or nonspam that might sneak through.

It is very important that no matter how you choose to deploy a spam filter, that you retain the ability to go back and fix problems such as misclassified email. In particular, shipment confirmations from companies like Amazon and ThinkGeek tend to look a lot like spam (actually, it's spammers trying to make spam that looks like Amazon and ThinkGeek shipping confirmations). That's why it's strongly recommended that you file spam into a folder that can be checked at leisure, boredom, or when you think you might have missed something important, rather than simply deleting it.

Connecting the Example Filter into your Mail System

Connecting your spam filter to your email system is a nontrivial subject because there are so many different mail delivery systems out there. There's no good general-case solution, and it's not possible to give a single set of instructions that work for everyone.

However, your CRM114 kit will contain a file named `CRM114_Mailfilter_HOWTO.txt` . This document contains the most up-to-date information that has been submitted on the subject of how to connect a mail filter to the ever-changing world of mail delivery agents.

There may also be one or more files in the CRM114 kit with names ending in `.recipe` . These are `procmail` recipes showing how to integrate a CRM114-based filter into your `procmail.rc` control file. If you have installed and are using `procmail` successfully, the `.recipe` files will give you a very good idea of how to proceed.

Maximum Message Length

A real issue for mail filter integration is where one draws the line between “really big but good messages” and “DoS messages” (“DoS” being Denial of Service). Many mail systems set the upper bound on any one email message's size at 1 to 4 megabytes; some are as low as 40 kilobytes, and some have no limit at all and will accept a message of any length. CRM114 has a default anti-DoS size limit of 8 megabytes; this can be altered on the command line to any 32-bit value, but that's not recommended. Instead, use **`head(1)`** to send CRM114 only the first 64Kbytes or so of an incoming message. For an example of how to do this, look in the CRM114 kit `.recipe` files.

There is no real consensus on what a good message size limit is; however, most email filters that use anti-DoS length limiting have found that filtering accuracy is basically undiminished if the filter sees the first 32 to 64Kbytes of the message, and there is only a small impact on accuracy if the filter is limited to the first 8Kbytes. However, the filtering time is at least linear in the length of the text processed- a series of very long messages may well slow down a mail server to the point where the server is effectively DoSed and no mail moves in or out of the site. For this reason, a filter limit of 64Kbytes to 100Kbytes is both reasonable and prudent; the filter makes the good/spam decision on the first 64 Kbytes of the incoming message.

Adding an optional MIME Normalizer

Some spam filtering users find that adding a MIME decoder to revert messages encoded as MIME objects can be useful. Typically, these encodings are using BASE64 encoding to break a word-matching tokenizer, HTML comments to break up spammish words, and the occasional rendering trick. Additionally, there are well-meaning packages (such as some web-based groupware packages) that fail to properly send plain ASCII as plain ASCII; this is a bug to be worked around. Lastly, some mailers use bizarre encodings (such as the Russian Cyrillic KOI-8) when the content is actually Latin-1. Converting these different encodings into something that a spam filter can work with is the purpose of a *mime normalizer.*

There are a number of quality mime normalizers available for free download; for the sake of example, we will use Jaakko Hyvätti's `normalizemime` program. Like CRM114, `normalizemime` is open source software; you download source and compile it locally.

Most mime normalizers run as separate programs, not as libraries. This is very easy to integrate with a CRM114 program via a SYSCALL.

Assuming you have now downloaded and installed `normalizemime`, we can SYSCALL to it in just one line. Because we want our whitelist and blacklist functions to operate on the normalized result, the SYSCALL has to happen before the call to `:triggerword_check:`.

This is the new main program:

```
#!/usr/bin/crm114
{
      isolate (:conclusion:) /unknown/
      syscall (:*:_dw:) (:_dw:) /normalizemime /
      call /:triggerword_check:/
      call /:dispatch_result:/
      call /:classify_check:/
      call /:dispatch_result:/
      output / ERROR – this program should never get to here! \n/
      exit /99/
}
```

and in the case of our previous examples, the function is identical because the example testing programs don't use any MIME encoding.

Section 7: Conclusions

There you have it; you now have the complete story on how to use CRM114 to write some seriously useful programs.

CRM114 isn't for everyone (or every problem), but as one very early version of the documentation said "`CRM114 is grep bitten by a radioactive spider`", because LEARN and CLASSIFY can pattern-match on ideas and common ways of expression rather than simple **`grep-`**able character strings. This is a very simple and useful form of artificial intelligence – AI.

That's the crux of why you will want CRM114 in your programming tool kit– CRM114's classifiers amount to "Artificial Intelligence In A Spray Can"; you can easily apply pattern-recognition AI to text based problems without writing any AI code.

Enjoy.

-Bill Yerazunis

“Klaatu! Barada! Nnnn ... nnnn... necktie?”

- Bruce Campbell, as Ash, in Army of Darkness.

Appendix A: The CRM114 Quick Reference

This appendix is based on the CRM114 Quick Reference Card, release 20061000 codename “BlameRobert”. You are encouraged to make photocopies of this section quick reference card for easy access.

The Command Line

Invoke as 'crm whatever' or use '

```
#!/usr/bin/crm
```

as the first line of a script file containing the program text.

-d N - run N cycles, then drop into debugger. If no N, debug immediately

-e - no environment variables imported

-E N - reserve exit codes zero through N for the user program; CRM114 engine error codes will start at N+1 with one unique code per fatal error. (if N is zero or unset, use system-defined EXIT_SUCCESS and EXIT_FAILURE)

-h - print help text

-l N - print a program listing – N from 1 to 5 changing detail level, 0 for no listing

-p - generate an execution-time-spent profile on exit

-P N - max program lines

-q m - math mode (0,1 = alg/RPN only in EVAL, 2,3 = alg/RPN everywhere)

-s N - new feature file (.css) size is N (default 1 meg+1 feature slots)

-S N - new feature file (.css) size is N rounded up to 2^I+1 feature slots

-t - user trace output, with -l matching line numbers

-T - implementors trace output (only for the masochistic!)

-u dir - chdir to directory dir before starting execution

-v - print CRM114 version identification and exit.

-w N - max data window (bytes, default 8 megs)

-- - signals the end of CRM114 flags; prior flags are not seen by the user program; subsequent args are not processed by CRM114.

--foo - creates the user variable **:foo:** with the value SET. Note that command line arguments have the surrounding colons removed.

--x=y - creates the user variable **:x:** with the value y. Again, you don't need colons around the variable name.

-{ stmts } - execute the statements inside the {} brackets.

Absent the **-{ program }** flag, the first arg is taken to be the name of a file containing a CRM114 program, subsequent args are merely supplied as **:_argN:** values. Use single quotes around command-line programs **'-{ like this }'** to prevent the shell from doing odd things to your command-line programs.

CRM114 can be directly invoked by the shell if the first line of your program file uses the shell standard, as in:

```
#! /usr/bin/crm
```

You can use CRM114 flags on the shell-standard invocation line, and hide them with **'--'** from the program itself; **'--'** incidentally prevents the invoking user from changing any CRM114 invocation flags.

Flags should be located after any positional variables on the command line. Flags are visible inside the program as **:_argN:** variables, so you can create your own flags for your own programs (separate CRM114 and user flags with '--').

Examples:

```
./foo.crm bar mugga < baz  -t -w 150000      <--- Use this
./foo.crm -t -w 1500000 -- bar < baz mugga   <--- or this
./foo.crm -t -w 150000 bar < baz mugga      <--- NOT like this
```

You can put a list of user-settable vars on the `'#!/usr/bin/crm'` invocation line. CRM114 will print these out when a program is invoked from the command line (e.g. **`./myprog.crm -h`** , not **`crm myprog.crm -h`** with the **`-h`** (for help) flag. (note that this works ONLY on bash on Linux- *BSD and Solaris have a different bash interpretation and this doesn't work. Don't use this in programs that need to be portable)

Example:

```
#!/usr/bin/crm  -( var1 var2=A var2=B var2=C )
```

- allows only **`var1`** and **`var2`** be set on the command line (without colons). If a variable is not assigned a value, the user can set any value desired. If the variable is equated to a set of values, those are the only values allowed.

```
#!/usr/bin/crm  -( var1 var2=foo )  --
```

- allows **`var1`** to be set to any value, **`var2`** may only be set to either "foo" or not at all, and no other variables may be set nor may invocation flags be changed (because of the trailing "**`--`**"). Since "**`--`**" also blocks **`-h`** for help, such programs should provide their own help facility.

Variables

Variable names and locations start with a : , end with a : , and may contain only characters that have ink (i.e. the `[:graph:]` class) with few exceptions (among them, a variable name may not contain an embedded colon).

Examples **`:here:`** , **`:ThErE:`**, **`:every-where_0123+45%6789:`** , **`:this_is_a_very_very_long_var_name_that_does_not_tell_us_much:`** .

Built in variables:

`:_nl:` - newline
`:_ht:` - horizontal tab

:_bs: - backspace
:_sl: - a slash
:_sc: - a semicolon
:_cd: - the call depth of the currently executing routine
:_cs: - the current statement number
:_arg0: thru **:_argN:** - command-line args, including _all_ flags
:_argc: - how many command line arguments there were
:_pos0: thru **:_posN:** - positional args ('-' or '--' args deleted)
:_posc: - how many positional arguments there were
:_pos_str: - all positional arguments concatenated
:_env_whatever: - environment value 'whatever'
:_env_string: - all environmental arguments concatenated
:_crm_version: - the version of the CRM system
:_pgm_hash: - hash of the current program - for version verification
:_pid: - the current process's PID
:_ppid: - the PID of the parent of the current process
:_dw: - the current data window contents (usually the default arg)

Variable Expansion

Variables are expanded by the **:*** var-expansion operator, e.g. **:*:_nl:** expands to a newline character, and the **:+** var-indirection operator. Uninitialized vars evaluate to their text name (and the colons stay). You can (if necessary) redefine some but not all such vars.

You can also use the standard constant C '\' characters, such as **\n** for newline, as well as escaped hexadecimal and octal characters like **\xHH** and **\oOOO** but these are constants, not variables, and cannot be redefined.

Depending on the value of "math mode" (flag **-q**). you can also use **:#:string_or_var:** to get the length of a string, and **:@:string_or_var:** to do basic mathematics and inequality testing, either only in EVALs or for all var-expanded expressions. See "Sequence of Evaluation" below for more details.

Default Program Behavior

Default behavior is to read all of standard input till EOF into the default data window

(which is named **:_dw:**), then execute the program (this is overridden if first executable statement is a WINDOW statement). Variables don't get their own storage unless you ISOLATE them (see below), instead variables are start/length pairs indexing into the default data window. Thus, ALTERing an unISOLATEd variable changes the value of the default data buffer itself. This is a great power,
so use it only for good, and never for evil.

CRM114 Statement Syntax

\ - '\\' is the string-text escape character. You only need to escape the literal representation of closing delimiters inside var-expanded arguments. You can use the classic C/C++ \\-escapes, such as **\n, \r, \t, \a, \b, \v, \f, \0,** and also **\xHH** and **\oOOO** for hex and octal characters, respectively. A '\\' as the last character of a line means the next line is just a continuation of this one. A \\-escape that isn't recognized as something special isn't an error; you may optionally escape any of these delimiters: **>)] } ; / # ** and get just that character. A '\\' anywhere else is just a literal backslash, so the regex **([abc])\1** is written just that way; there is no need to double-backslash the **\1** (although it will work if you do). This is because the first backslash escapes the second backslash, so only one backslash is "seen" at runtime.

# this is a comment

# and this too \# - A comment is not a piece of preprocessor sugar- it is a statement, can only occur at “statement level”, and ends at the newline or at **\#** . Putting a # inside a delimited string (as in the statement **OUTPUT / HELLO # WORLD #\n/** does not cause a comment at that point..

insert *filename* - inserts the file verbatim at this line at compile time. If the file can't be INSERTed, a system-generated FAULT statement is inserted. Use a TRAP to catch this fault if you want to allow program execution to continue without the missing INSERT file.

; - statement separator - unless it's inside delimiters all semicolons must be escaped as **\;** or else it will mark the end of the statement.

{ and **}** - start and end blocks of statements. Curly braces must always be either inside delimiters or used with a \\-escape or else they will mark the start or end of a block.

noop - NO-OP statement. A blank line is also a NOOP.

:label: - define a GOTOable label

:label: ***(:arg:)*** - define a CALLable label. The args in the CALL statement are concatenated and put into the freshly ISOLATEd var *:arg:*

(:arg:) - var-expanded variable name to receive the caller's arguments (usually a MATCH is then done to put locally convenient labels on the args).

accept - writes the current data window to standard output; program execution continues.

alius - if the last bracket-group succeeded, ALIUS skips to end of **{}** block (a skip, not a FAIL); if the prior group FAILed, ALIUS does nothing. Thus, ALIUS is both an ELSE clause and a CASE statement.

alter ***(:var:) /new-val/*** - surgically change value of var to newval

(:var:) - var to change (var-expanded)

/new-val/ - value to change to (var-expanded)

call ***/:entrypoint_label:/ [:arg1: :arg2:...] (:retarg:)*** - do a subroutine call on the specified (var-expanded) entrypoint label. Note that the called routine shares all variables (including the data window **:_dw:**). Return is accomplished with the RETURN statement.

/:entrypoint_label:/ - the location to call

[:arg1: :arg2: ...] - var-expanded list of args to call. These are concatenated and supplied to the called routine as a single ISOLATEd string variable, to be used as desired (usually a MATCH parses the arglist as desired). These arguments are NOT modified (they are read only, but if a variable name is included, then the variable's contents can be changed).

(:retarg:) - this variable gets the returned value from the routine called (if it returns anything). If it had a previous value, that value is overwritten on return. Like the calling arguments, you can use a raw value here, or have the subroutine return one or more variable names containing the result.

classify ***<flags> (:c1:...|..:cN:) (:stats:) [:in:] /word-pat/ /pR_offset/***

- compare the statistics of the current data window buffer with class files c1...cN

<nocase> - ignore case in word-pat, does not change case in hash (use **tr()** to do that on **:in:** if you want it)

<microgroom> - enable the microgroomer to purge less-important information automatically whenever the statistics file gets too crowded. Downside: this disables certain optimizations that can speed classification.

<osb> - use orthogonal sparse bigram (OSB) features and Markovian classification instead of Markovian SBPH features. OSB uses a subset of SBPH features with about 1/4 the memory and disk needs, and about 4x the speed of full Markovian, with basically the same accuracy.

`<unique>` - use only the presence or absence of a feature, rather than the count of the number of occurrences. This speeds things up and often improves accuracy.

`<osbf>` - use the Fidelis Confidence Factor local probability generator. The file format is not compatible with the default, but with single sided threshold training (typically pR of 10-30) achieves the best accuracy yet. OSBF may not converge on all possible training sets. Uses .csf statistics files, which are not compatible with .css files.

<winnow> - use the Winnow non statistical classifier and the OSB front end feature generator. Winnow uses .cow files, which are not compatible with the .css files for the Markovian (default) and OSB classifiers.

<correlate> - use the full correlative matcher. Very slow, but capable of matching stemmed words in any language and of matching binary files. Uses plain text statistics files, and is not compatible with any other statistics file format.

`<hyperspace>` - use the hyperspace matcher. This is a very fast and accurate, but still experimental and poorly supported classifier. Instead of intermingling the features found in each of the example files, hyperspatial classification keeps each set of features separate, and uses a distance measurement from each learned example to the unknown text. The distances are weighted as if there was a light source at the location each of the known texts, and the class that provides a greater level of radiance on the unknown text's position in hyperspace is the class that “wins” the classification. Hyperspace classification uses the OSB feature set.

`<entropy>` - use the entropic classifier. Not quite as accurate as OSB or OSBF or Hyperspace for minimum total errors, but works better when the cost of one kind of error is much greater than the cost of the other kind of error. Recommended to use with **`<unique>`** and **`<crosslink>`**

`(:c1: ...` - file or files to consider "success" files. The CLASSIFY succeeds if these files as a group match best; if not, the CLASSIFY does a FAIL.

`|` - (vertical bar) - optional separator. Spaces on each side of the " | " are required. The vertical bar separates “success” files from “fail” files in a CLASSIFY match.

`.... :cN:)` - optional files to the right of " | "are considered as a group to "fail". If the statement fails, execution skips to end of enclosing {..} block, which exits with a FAIL status (see ALIUS for why this is useful).

`(:stats:)` - optional var that will be surgically changed to contain a formatted matching summary

`[:in:]` - restrict statistical measure to the string inside **`:in:`**

`[:in: :start: :len: /regex/ ]` - restrict statistical measure to within the start/length pair and/or the restriction regex given. Multiple start/length pairs and regexes may be pipelined.

/word-pat/ - regex to describe what a parseable word is. The default regex is `[[:graph:]]+`

/pR_offset/ - *OSBF ONLY*: By default, a classify succeeds for pR > 0. With this optional parameter the success/failure decision point can be changed from the default 0 to what you specify. If given, the pR in 'stats' will be printed in the form pR/pR_offset.

debug – drop immediately into the interactive debugger.

eval *(:result:) /instring/* - repeatedly evaluates /instring/ until it ceases to change, then surgically places that result as the value of :result: . If the instring uses math evaluation (see section below on math operations) and the evaluation has an inequality test, (>, < or =) then if the inequality fails, the EVAL will FAIL to the end of block. If the evaluation has a numeric fault (e.g. divide-by-zero) the EVAL will do a TRAPpable FAULT.

exit /:exitcode:/ - ends program execution. If supplied, the return value is converted to an 8-bit integer and returned as the exit code of the CRM114 program.

/:exitcode:/ - variable to be converted to an integer and returned. If no exit code is supplied, the default exit code value is 0.

fail - skips down to end of the current **{ }** block and causes that block to exit with a FAIL status (see ALIUS for why this is useful)

fault */faultstr/* - forces a FAULT with the given string as the reason.

/faultstr/ - the val-expanded fault reason string

goto /:label:/ - unconditional branch (you can use a var-expanded variable as the goal, e.g. **/:*:there:/** is legal)

hash (:result:) /input/ - compute a fast 32-bit hash of the /input/, and ALTER **:result:** to the ASCII representation of the hexadecimal hash value. HASH is not warranted to be constant across major releases of CRM114, nor is it cryptographically secure.

(:result:) - value that gets results

/input/ - string to be hashed (can contain expanded **`:*:vars:`** , defaults to the data window **`:_dw:`**)

intersect ***(:out:) [:var1: :var2: ...]*** - makes :out: contain the part of the data window that is the intersection of **`:var1 :var2:`** ... ISOLATEd vars are ignored. This only resets the value of the captured **`:out:`** variable, and does NOT alter any text in the data window.

isolate ***(:var:) /initial-value/*** - isolates **`:var:`** from all other variables so subsequent changes to this variable don't change any other variable (though they may change the value of any variable subsequently set inside of this variable). If the variable was already ISOLATED, the variable will stay isolated but it will surgically alter the value if a **`/value/`** is given.

(:var:) - name of ISOLATEd variable (var-expanded)

`<default>` - only create and set **`:var:`** if it didn't exist before. Ideal for setting defaults.

/initial-value/ - optional initial value for :var: (var-expanded). If no value is supplied, the previous value is retained or copied from the prior value.

input ***<flags> (:result:) [:filename:]*** - read in the content of filename. If no filename is given, then read `stdin`

`<byline>` - read one line only, without waiting for EOF (useful only on `stdin`)

(:result:) - var that gets the input value (surgical overwrite).

[:filename:] - the file to read. The first blank-delimited word is taken and var-expanded; the result is the filename, even if it includes embedded spaces. Default is to read `stdin`.

[:filename: offset len] - optionally, move to *`offset`* in the file, and read *`len`* bytes. Offset and length are individually blank-delimited, and var-expanded with mathematics enabled. If *`len`* is unspecified, the read extends to EOF or buffer limit.

learn ***<flags> (:class:) [:in:] /word-pat/*** - learn the statistics of the **:in:** variable (or the input window if no variable is given) as a textual example of class **:class:**

<nocase> - ignore case in matching word-pat (does not ignore case in the actual text - use **tr()** to do that on **:in:** if you want it)

<refute> - flag this is as an anti-example of this class- unlearn it!

<microgroom> - enable the microgroomer to purge less-important information automatically whenever the statistics file gets too crowded. However, this disables other optimizations that can speed up the classifier.

<osb> - use orthogonal sparse bigram (OSB) features and Markovian classification instead of Markovian SBPH features. OSB uses a subset of SBPH features with about 1/4 the memory and disk needs, and about 4x the speed of full Markovian

<osbf> - use the Fidelis Confidence Factor (EDDC) local probability generator. This format is not compatible with the default, but with single sided threshold training (typically pR of 10-30) achieves the best accuracy yet. However, OSBF is not guaranteed to converge.

<winnow> - use the Winnow non statistical classifier and the OSB front end feature generator. Winnow uses .cow files, which are not compatible with the .css files for the Markovian (default) and OSB classifiers.

<correlate> - use the full correlative matcher. Very slow, but capable of matching stemmed words in any language and of matching binary files. Correlative matching does not tokenize, and so you don't need to supply it with a word-pat. regex

<hyperspace> - use the hyperspace matcher. This is a very fast and accurate, but still experimental and poorly supported classifier. Instead of intermingling the features found in each of the example files, hyperspatial classification keeps each set of features separate, and uses a distance measurement from each learned example to the unknown text. The distances are weighted as if there was a light source at the location each of the known texts, and the class that provides a greater level of radiance on the unknown text's position in hyperspace is the class that "wins" the classification. Hyperspace classification uses the OSB feature set.

`<entropy>` - use the entropic classifier. Not quite as accurate as OSB or OSBF or Hyperspace for minimum total errors, but works better when the cost of one kind of error is much greater than the cost of the other kind of error. Recommended to use with **`<unique>`** and **`<crosslink>`**

`(:class:)` - name of file holding hashed results; the nominal file extension for Markovian and OSB classifiers is **`.css`**

`[:in:]` - captured var containing the text to be learned (if omitted, the full contents of the data window is used)

`[:in: :start: :len: /:regex:/]` - restrict learning to the start/length pair and/or the restriction regex given. Multiple start/length pairs and regexes may be pipelined.

`/word-pat/` - regex that defines a "word". Things that aren't "words" are ignored. Default is `/[[:graph:]]+/`

`liaf` - skips up to start of the current **`{}`** block; this is the opposite of FAIL. (mnemonic: LIAF is FAIL spelled backwards)

`match` *`<flags> (:var1: ...) [:in:] /regex/`* - Attempt to match the given regex; if the match succeeds, the variables are bound; if the match fails, the program skips to the closing **`}`** of this block

`<absent>` - statement succeeds if the match not present

`<nocase>` - ignore case when matching

`<literal>` - No special characters in regex (only supported with TREregex, not GNUregex.)

`<fromstart>` - start match at start of the **`[:in:]`** var (this is the default starting location for the match – use it to emphasize start-of-var in loops containing other flagged matches)

`<fromcurrent>` - start match at start of previous successful match on the **`[:in:]`** var. (note- there isn't any “stack”; you cannot go back to prior matches)

`<fromnext>` - start match at one character past the start of the previous successful match on the **`[:in:]`** var

`<fromend>` - start match at one character past the end of previous successful match on this **`[:in:]`** var

`<newend>` - require the match to end after end of previous successful match on this **`[:in:]`** var. This is done iteratively, moving the start of attempted matching one character at a time until an acceptable result is found (or the match fails).

`<backwards>` - search backward in the **`[:in:]`** variable from the last successful match. This is done iteratively, moving the start of the attempted matching one character backward at a time until an acceptable result is found (or the match fails)

`<nomultiline>` - don't allow this match to span lines. As a side effect, ^ will match at the start of each line, and **`$`** will match at the end of each line.

`(:var1: ...)` - optional result vars. The first var gets the text matched by the full regex. The second, third, etc.vars get each subsequent parenthesized subexpression, in left-to-right order of the subexpression's left parentheses. These are "captures", not ALTERs, so text overlapping prior **`:var:`** values is left unchanged; the old string values stay in their places and only the variable pointers move to point to the newly matched start and lengths. It is suggested as a convention to use the variable **`::`** as a “placeholder” variable for any value you are discarding.

`[:in:]` - search only in the variable specified; if omitted, the default **`:_dw:`** (the full input data window) is used

`[:in: :start: :len:]` - search in the *`:in:`* input var, limiting the area searched to the substring from *`:start:`* to *`:len:`* (zero-origin counted). Both *`:start:`* and *`:len:`* are evaluated with full math evaluation enabled.

`[:in: :start: :len: /:regex:/]` - restrict matching to within the

start/length pair and/or the restriction regex given. Multiple start/length pairs and regexes may be pipelined.

`/regex/` - POSIX regex (with **`\`** escapes as needed)

`output` *`<flags> [filename] /output-text/`* - output an arbitrary string with captured values var-expanded.

`<append>` - append to the file (otherwise, the file is overwritten from character 0, which causes the previous contents of the file to be truncated away by the file system).

`[:filename:]` - the file to write. The first blank-delimited word is taken and var-expanded; the result is the filename, even if it includes embedded spaces. Default output is to **`stdout`**. The special file **`stderr`** is also acceptable- this is the standard error pipe, not a file named *`stderr`*.

`[:filename: offset len]` - optionally, move to *`offset`* in the file, and maximum write *`len`* bytes. *`Offset`* and *`len`* are individually blank-delimited, and var-expanded with mathematics enabled. If *`len`* is unspecified, the write is the length of the expansion of *`/output-text/`*

`/output-text/` - string to output (var-expanded)

`return` ***`/returnval/`*** - return from a CALL. Note that since CALL executes in shared space with the caller, all changes made by the CALLed routine are shared with the caller.

`/returnval/` - this (var-expanded) value is returned to the caller (or if the caller doesn't accept return values, it's discarded).

`syscall` ***`<flags> (:in:) (:out:) (:status:) /command_or_label/`*** - execute a shell command or fork to the specified label. This happens in a `fork()`-style copy of the current CRM114 environment; there is no communication with the main program except via the ***`:in:`***, ***`:out:`***, and ***:status:*** vars. Output past the maximum buffer length is discarded unless you **`<keep>`** the process around for multiple reuses.

`<keep>` - don't send an EOF after feeding the full input buffer to the new process (this will usually keep the syscalled process around). Later syscalls with the same **`:status:`** var will continue feeding to and reading from the kept process. This is designed for interaction with “conversational” programs, like **`ssh, ftp, nc, bash`**, or database managers.

`<async>` - don't wait for process to output an EOF; just grab what's available in the process's output pipe and proceed (default limit per syscall is 256 Kbytes). The syscalled process then runs to completion independently and asynchronously; any later output is discarded. (This is "fire and forget" mode, and is mutually exclusive with **`<keep>`**.)

`(:in:)` - var-expanded string to feed to command as input (can be null if you don't want to send the process something.) You must specify this if you want to specify an ***:out:*** variable.

`(:out:)` - var-expanded varname to place results into (must pre-exist, can be null if you don't want to read the process's output (yet, or at all). Limit per syscall is 256 Kbytes. You must specify an ***:out:*** var if you want to use the *`:status:`* variable. The alteration of the ***:out:*** variable is surgical; if it overlaps any other variable, that variable will also see the alteration.

`(:status:)` - if you want to keep a minion process around, or catch the exit status of the process, specify a variable name here. The minion process's PID and pipes will be stored here (as human-readable strings). The program can access the process again with another SYSCALL by using this var again. When the process exits, its exit code will be surgically stored here (unless you specified **`<async>`**)

`/command_or_label/` - the command or entry-point you want to run. This argument is var-expanded; if the first word is a **`:label:`**, the fork begins execution at the label. If the first word is not a **`:label:`**, then the entire string is handed off to the shell to be executed as a shell command. Redirection with <, >, and >> is allowed for both `bash` commands and for **`:label:`** forked

minions, but no spaces can exist between the redirection character and the redirection target name.

translate <flags> ***[:src_var:] (:dest_var:) /from_charset/ /to_charset/*** - bytewise translation of characters found in the ***from_charset*** to the corresponding characters in the ***to_charset***. If no to_charset is supplied, it deletes characters in the from_charset.

<unique> - repeated sequential copies of the same char in ***from_charset*** are replaced by a single copy, then translated.

<literal> - ***from_charset*** and ***to_charset*** are literal, so no var-expansion, range expansion, or inversion is performed

[:src_var:] - source of the text. Can be var-restricted; default is the default data window **:_dw:**

(:dest_var:) - where result of the translation will be placed; default is ***:_dw:***

/from_charset/ -var-expanded charset of characters to be translated from. Use hyphens for ranges like **a-e** which means **abcde** . Reversed ranges such as **e-a** meaning **edcba** work. (this is different than **tr())**. Set inversion as in **^a-z** mean all characters that aren't lower case characters works. Character duplication is not an error. To use a literal hyphen "-" as a literal character, make it any bu the first character; to use a literal caret make it any but the first character. ASCII \-escapes and **\xFF** hex characters work.

/to_charset/ - charset of characters to be translated to, with the same rules as the ***from_charset.*** Extra characters are ignored, and if not enough characters are supplied, the entire set supplied in ***to_charset*** is reused from the beginning in order (this is different than **tr()**

trap ***(:reason:) /trap_regex/*** - traps faults from both FAULT statements and program errors occurring anywhere in the preceding bracket-block. If no fault exists, TRAP does a SKIP to end of block. If there is a fault and the fault reason string matches the `/trap_regex/`, the fault is trapped, and execution continues with the line after the TRAP, otherwise the fault is passed up to the next surrounding trapped bracket block.

`(:reason:)` - a freshly ISOLATEd variable that contains the fault message that caused this FAULT. If it was a user fault, this is the text the user supplied in the FAULT statement.

`/trap_regex/` - the regex that determines what kind of faults this TRAP will accept. Putting a wildcard here (e.g. **`/.*/`** means that ALL trappable faults will be trapped.

`union` ***`(:out:) [:var1: :var2: ...]`*** - makes **`:out:`** contain the union of the data window segments that contains *`var1, var2`*`...` plus any intervening text as well. Any ISOLATEd var is ignored. This is non-surgical, and does not alter the data window.

`window` ***`<flags> (:w_var:) (:s_var:) /cut-regex/ /add-regex/`*** - slide a window through the ***`:w_var:`***, making successive cuts and appends. Specifically, this statement deletes everything from the start of the **`:w_var:`** up to and including the *`/cut-regex/`* from ***`:w_var:`*** (***`:w_var:`*** defaults to **`:_dw:`**, the default data window), and then appends more text from ***`:s-var:`*** or `stdin` (default `stdin`) until we find *`/add-regex/`*. Both the cut-regex and the add-regex should match only the edge marker rather than being all-inclusive.

`<nocase>` - ignore case when matching cut- and add- regexes

`<bychar>` - (default) read one char at a time and check input for add-regex every character, so we never read "too much" from `stdin`. This is the “most compatible” mode for operating within another scripted language context. It never reads too many characters from `stdin`, but it's much slower than **`<bychunk>`** or **`<byeof>`**

`<bychunk>` - reads as much data as available, then check with the regex. (unused characters are kept around for later WINDOW operations).

`<byeof>` - wait for EOF to check add-regex (unused characters are kept around for later WINDOW operations)

`<eofaccepts>` - accept an EOF as being a successful regex match (default is only a successful add-regex matches. CAUTION: this can cause rapid looping!)

`<eofretry>` - keep reading past an EOF; reset the stream and wait again for more input. (default is to FAIL on EOF. CAUTION: this can cause rapid looping!)

`(:w_var:)` - the windowed var -what var to cut from and then add to

`(:s_var:)` - the source var - what var to use for source (defaults to `stdin`) if you use a source var you must specify the windowed var. as well

`/cut-regex/` - var-expanded cut regex pattern. Everything up to and including the first match of this regex is deleted.

`/add-regex/` - var-expanded add pattern. if absent, reads till EOF. This pattern is a minimal match pattern, so if the pattern can match a zero-length string (say, /.*/), this can yield zero characters added. Use a pattern like /.+/ to prevent this.

If both the cut-regex and the add-regex are omitted, and this window statement is an executable no-op... except that if it's the first executable statement in the program, then the WINDOW statement configures CRM114 to not wait to read a anything from standard input before starting program execution.

Quick Summary of Regex Expressions.

A regex is a pattern match. Matches are, by default "first starting point that matches, then longest match possible that can fit".

`a` through **`z`**
`A` through **`Z`** - all match themselves

`0` through **`9`**

Most punctuation characters match themselves, but check below!

`.` - the 'period' char, matches any character

`*` - repeat preceding 0 or more times

`+` - repeat preceding 1 or more times

`?` - repeat preceding 0 or 1 time

`[abcde]` any one of the letters a, b, c, d, or e

`[a-q]` the letters a through q (just one of them)

`[a-eh-mqzt]` the letters a through e, plus h through m, plus q, z, and t

`[^xyz]` any one letter EXCEPT one of x, y, or z

`[^a-e]` any one letter EXCEPT one of a through e

`{n,m}` repetition count: match the preceding at least n and no more than m times (sadly, POSIX restricts this to a maximum of 255 repeats. Nested repeats like `(.{255}){10}` will work, but are very slow).

`[[:<:]]` matches at the start of a word (GNU regex only)

`[[:>:]]` matches the end of a word (GNU regex only)

`^` a carat as the first char of a match, matches the start of a line (ONLY in **`<nomultiline>`** matches; otherwise 'start of buffer')

`$` a dollar sign as last char of a match, matches at the end of a line (ONLY in **`<nomultiline>`** matches, otherwise 'end of buffer')

`.` (a period) matches any single character (except start-of-line or end of line "virtual characters", but it does match a newline).

`(match)` - the () go away, and the string that matched inside is available for capturing. Use \(and \) to match actual parentheses.

a|b match a _or_ b, such as foo|bar (multiple characters!). To get a shorter extent of ORing, use parentheses, e.g. /f(oo|ba)r/ matches "foor" or "fbar", but not foo or bar.

The following are other POSIX expressions, which mostly do what you'd guess they'd do from their names.

[[:alnum:]]
[[:alpha:]]
[[:blank:]] <-- space and tab only
[[:space:]] <-- "whitespace" (space, tab, vertical tab (^K), \n, \r, ..)
[[:cntrl:]]
[[:digit:]]
[[:lower:]]
[[:upper:]]
[[:graph:]] <-- any character that puts ink on paper or lights a pixel
[[:print:]] <-- any character that moves the "print head" or cursor.
[[:punct:]]
[[:xdigit:]]

TRE-only Regex Extensions

These expressions will work only in TRE, not in GNU regex. However, the default CRM114 install uses TRE, so they're pretty safe to use.

?, +?, ??** **-** repeat preceding just like **, **+**, and **?**, but instead of the longest match, these accept the shortest match that fits, given the already-selected start point of the regex.

\*N* - where N is an integer from 1 to 9 - matches the N'th parenthesized subexpression. You don't have to backslash-escape the backslash (e.g. write this as **\1** or as **\\1**, either will work)

\Q - start verbatim quoting - all following characters represent exactly themselves; no special characters are respected. This is only terminated by a **\E** or the end of the regex.

\E - end of verbatim quoting.

\< - start of a word (doesn't use up a character)

\> - end of a word (doesn't use up a character)

\d - a digit.

\D - not a digit

\s - a space

\S - not a space

\w - a word char (a-z, A-Z, 0-9, or _)

\W - not a word char

(?:some-regex) - parenthesize a subexpression, but <u>don't</u> capture a submatch for it.

(?inr-inr:regex) - Let you selectively turn on or off case independence, nomultiline, and right-associative (rather than the default left-associative) matching within various parts of a regex. These nest as well.

i - case independent matching. examples:

/(?i:abc)/	matches 'abc', 'AbC', 'ABC', etc...
/(?i:ABC(?-i:de)FGH)/	matches ABCdeFGH, abcdefgh, but not ABCdEFGH or ABCDEFGH

n - don't match newlines with wildcards such as **.*** or with anti-wildcards like **[^j-z]**. **-n** <u>allows</u> matching of newlines (this is slightly counterintuitive). e.g.:

/(?n:a.*z)/ matches 'abcxyz' but not

'abc
xyz'

/(?-n:a.*z)/ matches both (this does NOT override the <nomultiline> flag; <nomultiline> essentially "blocks" the searched text into chunks at each newline, and searches within each chunk separately, so the regex never bridges across chunks)

r - right-associate matching on **|** alternatives. This changes only sub-matches; never whether the match itself succeeds or fails. Normally, in alternative matches, if two alternatives yield the same length overall match, the one on the left is chosen. If you specify **?r:** then the one on the right is chosen. For example the regex **(a|ab)(c|bcd)(d*)** will match **abcd** with the submatches of **a**, **bcd**, and **<empty>** but **(?r:(a|ab)(c|bcd)(d*))** will match **abcd** with submatches of **ab**, **c**, and **d**. This is rather subtle, if you don't understand it at first, play with the regexes a bit to see if it becomes clearer.

Notes on the Sequence of Evaluation

By default, CRM114 supports string length and mathematical evaluation only in an EVAL statement, although it can be set to allow these in any place where a var-expanded variable is allowed (see the -q flag). The default value (zero) allows string length and math evaluation only in EVAL statements, and uses non-precedence (that is, strict left-to-right unless parentheses are used) algebraic notation. -q 1 uses RPN instead of algebraic, again allowing string length and math evaluation only in EVAL expressions. Modes 2 and 3 allow string length and math evaluation in <u>any</u> var-expanded expression, with non-precedence algebraic notation and RPN notation respectively.

Evaluation is always left-to-right; there is no precedence of operators beyond the sequential passes noted below. The evaluation is done in five sequential passes:

1) ****-constants like **\n**, **\o377** and **\x3F** are substituted in. You must use three digits for octal and two digits for hex. To write something that will literally appear as one of these constants, escape the backslash with another backslash, i.e. to output `\o075` use `\\o075`.

2) **:*:var:** variables are substituted (note the difference between a constant like **\n** and a variable like **:*:_nl:** here – constants are substituted first, then variables are substituted.)

3) **:+:var:** variables go through variable indirection. The indirect variable name is substituted once to get the real variable name, and then the real variable name is expanded to the value of the variable.

4) **:#:var:** string-length operations are performed. You don't have to expand a **:var:** first, you can take the string length directly, as in **:#:_dw:** to get the length of the default data window. Thus, you can take the length of a string that contains a **:** which would normally "end" the **:#:** operator .

5) **:@:expression:** mathematical expressions are performed; syntax is either RPN or non-precedence (parens required) algebraic notation. Embedded non-evaluated strings in a mathematical expression is currently a no-no. The allowed operators in a mathematical expression are: **+ - * / % > < = E F G X** . (E, F, G and X are "formatting operators", E, F, and G cause format-based rounding at that point in the computation,

and all four set the final output format). Input numbers may be expressed as integers, as decimals (that is, numbers with an included decimal point), as E-notation floating points (that is, an integer or a decimal followed immediately by an E and then a positive or negative exponent, or as a 32-bit hexadecimal constant like 0xFDA139C . Only **>, <**, and **=** set logical results; they also evaluate to 1 and 0 for continued chain operations - e.g.

```
((:*:a: > 3) + (:*:b: > 5) + (:*:c: > 9) > 2)
```

is true IFF any of the following is true

a > 3 and **b > 5**
a > 3 and **c > 9**
b > 5 and **c > 9**

Notes on Approximate REGEX Matching

The TRE regex engine (which is the default engine) supports approximate matching. The GNU engine does not support approximate matching.

Approximate matching is specified similarly to a "repetition count" in a regular regex, using brackets. This approximation applies to the previous parenthesized expression (again, just like repletion counts). You can specify maximum total changes, and how many inserts, deletes, and substitutions you wish to allow. The minimum-error match is found and reported, if it exists within the bounds you state. The basic syntax is:

```
(text-to-match){~[maxerrs] [#maxsubsts] [+maxinserts] [-maxdeletes]}
```

Note that the **~**(with an optional maxerr count) is required (that's how we know it's an approximate regex rather than just a rep-count); if you don't specify a maximum error count the match will always succeed and you will get the best match. If you do specify a maximum error count, the match will have at most that many errors.

Remember that you specify the changes to the text in the pattern necessary to make it match the text in the string being searched, not changes in the text to make it match the pattern. The only changes that are made are those in “non-special” characters (the approximate regex algorithm will never insert a **(.*)** into your regex; it might change an A to a Z, but adding or removing repeat counts, groupings, or alternate paths is not allowed)

You cannot use approximate regexes and backward references (like **\1**) in the same regex.

This is a limitation of in TRE at this point.

You can also use an inequality in addition to or instead of the basic approximation syntax above:

```
      (text-to-match){~[maxerrs] [basic-syntax] [nI + mD + oS < K] }
```

where **n**, **m**, and **o** are the costs per insertion, deletion, and substitution respectively, **I**, **D**, and **S** are indicators to tell which cost goes with which kind of error, and **K** is the total cost of the errors; the cost of the errors is always strictly less than **K**.

Here are some examples of approximate matching.

(foobar) - exactly matches "foobar"

(foobar){~} - finds the closest match to "foobar", with the minimum number of inserts, deletes, and substitutions. This match always succeeds, as six substitutions or additions is always enough to turn any string into one that contains 'foobar'.

(foobar){~3} - finds the closest match to "foobar", with no more than 3 inserts, deletes, or substitutions.

(foobar){~2 +2 -1 #1) - find the closest match to "foobar", with at most two errors total, and at most two inserts, one delete, and one substitution.

(foobar){~4 #1 1i + 2d < 5 } - find the closest match to "foobar", with at most four errors total, at most one substitution, and with the number of insertions plus 2x the number of deletions less than 5.

(foo){~1}(bar){~1) - find the closest match to "foobar", with at most one error in the "foo" and one error in the "bar".

Notes on Classifiers

CRM114 allows the user a whole gamut of different classification algorithms, and various tunings on classifications. The default classifier is a Markovian classifier that attempts to model the language as a Markov Random Field with site size of 5 (in plain language, the Markovian classifier looks at each word in the context of a window 5 words long; words within that window are considered "directly related" and are used to generate local probabilities. Words outside that 5-word window are not considered in relation to each word, but get considered when the window slides over to them).

The Markovian classifier is quite fast; more than fast enough for a single user or even a small office. Filtering speed varies- with no optimization and overflow repair (that is, with **`<microgroom>`** enabled) filtering speed is usually in excess of what a fractional T1 line can downlink.

The Markovian filter can be sped up considerably by turning off overflow safeguarding by not using **`<microgroom>`**; this optimization speeds up learning significantly, but it means that learning is unsafe. System operators must instead manually monitor the fullness of the .css files and either manually groom them or expand them as required (or a script must be used to automate this maintenance, which can be done "in flight").

A yet faster filter is the OSB filter, based on orthogonal sparse bigrams. OSB is natively about 4x faster than full Markovian, but loses some of this advantage if overflow safeguarding (no **`<microgroom>`**) is used. OSB is almost as accurate as Markovian if disk space is unlimited, and more accurate than Markovian if disk space is very limited.

A very new filter with excellent statistics is the Winnow filter. Winnow is a non-statistical method that uses the OSB front end feature generator. Winnow is different than the statistical filters, in that it requires both positive training and negative training to work, but then it works <u>very</u> well.

With Winnow, you don't just train errors into the correct class (i.e .in emulation of an SVM). Instead, you set a "thick threshold" (usually about +/- 0.2 in the pR scale), and any positive class that doesn't get a per-correct-file score of at least 0.2 pR gets trained as a positive example. Symmetrically, any negative class and negative example that doesn't get below -0.2 of pR needs to be trained as a negative example (that is, using the flags **`<winnow refute>`**.)

This means that with Winnow, on an error you train one or both files. Even if the classifier gives the correct overall result, if the per-file pR values are inside the -0.2 <= per_file_pR <= 0.2 thick-threshold, you may have to train one or both files as well. (these per-file pR values are in the statistics output variable).

The slowest classifier is the correlative classifier. This classifier is based on a full NxM correlation of the unknown text against each of the known text corpora. It's very slow (perhaps 100x slower than Markovian) but is capable of classifying texts containing stemmed words, of texts composed of binary files, and texts that cannot be reasonably "tokenized".   The **`<correlate>`** filter should be considered perpetually an experimental feature, and it is not as well characterized as the Markovian or OSB filters.

Another classifier is the OSBF (OSB with Fidelis mods) filter. The good news is it's even more accurate, faster, and needs fewer buckets than any of the other filters. The bad news is that it's voodoo; some parts of it seem to make little mathematical sense. But the reality is that it works very, Very, VERY well. It's incompatible with any of the other filters (uses .cfc files).

The newest classifier is the Hyperspace classifier. The hyperspace classifier isn't statistical at all- instead, it positions each separate example document as a light source in a high-dimensional hyperspace and puts the unknown document as an observer in that hyperspace. Whichever class illuminates the observer more brightly is the "correct" class. Hyperspace classification is *very* fast, and as accurate as OSB, while using only a few hundred Kbytes for statistics. The hyperspace classifier is still somewhat experimental, and the file format for hyperspace data storage is subject to incompatible upward changes so don't throw away your text files if you use hyperspace.

Overall Language Notes

Here's how to remember what goes where in the CRM114 language.

Unlike most computer languages, CRM114 uses inflection (or declension) rather than position to describe what role each part of a statement plays. The declensions are marked by the delimiters- the `/`, `(` and `)`, `<` and `>`, and `[` and `]`.

By and large, you can mix up the arguments to each kind of statement without changing their meaning. Only the ACTION needs to be first. Other parts of the statement can occur in any order, save that multiple **`(paren_args)`** and **`/pattern_args/`** must stay in their nominal order but can go anywhere in the statement. They do not need to be consecutive.

The parts of a CRM114 statement are:

ACTION - the verb. This is at the start of the statement.

`/pattern/` - the overall pattern the verb should use, analogous to the "noun subject" of the statement. Although this is often a regex, do <u>not</u> think of it that way, because there are common statements (OUTPUT, GOTO, RETURN, END, etc) where it's a textual variable.

`<flags>` - modifies how the ACTION does the work. You'd call these "adverbs" in human languages.

`(vars)` - what variables to use as adjuncts in the action (what would be called the “direct objects"). These can get changed when the action happens.

`[limited-to]` - where the action is allowed to take place (think of it as the "indirect object"). Generally these are not directly changed by the action.

“You are in a maze of twisty little passages, all alike”

- Zork

Appendix B: Getting and Installing CRM114

Here's how to get and install CRM114.

Spam filtering is shooting at a moving target- filters that were perfectly effective at one point in time may be compromised within weeks to months, and hopelessly outclassed within a year. Because CRM114 is typically used as a spam filter, CRM114 will often need to be revised in minor ways.

In general, CRM114 versions will be named in a three-part system such as:

```
  crm114-releasedate.codename
```

The release date is important; generally speaking later releases will have fewer bugs and better accuracy. Release date coding is four digits of year, then two digits of month, then two digits of day, so an alphanumeric sort will always sort CRM114 versions correctly.

The “codename” section of a release is simply there to improve human recognition; people remember things like “Clockwork Orange” and “San Andreas” better than a string of nearly random digits. Another use of the “codename” of a release is to pay particular homage to a person or thing that had major impact on that release, either through code submission, wisdom contribution, or simply pointing out a particularly juicy bug.

Step 1: Downloading.

Get yourself a copy of a CRM114 kit. The kits can always be found by visiting the CRM114 homepage at:

```
      http://crm114.sourceforge.net
```

You will need at least the statically-linked binary kit (compiled to run on any i386 or better Linux box); for best performance it is suggested you get the source kit and compile it on the same processor you will be running CRM114 on. If you do not have root authority on the machine you will be running CRM114 on, it is suggested you stay with the statically linked binaries (this is because the recommended "TRE" REGEX library requires either root to install, or major workaround mojo).

The kits are named:

```
    crm114-<version>.i386.tar.gz   (statically linked binaries)
and
    crm114-<version>.src.tar.gz    (complete source code + tests)
and
    crm114-<version>.i386.rpm      (statically linked .rpm package)
```

These kit .gz files are fairly small; usually around one megabyte, so they will download quickly.

Step 2: Setting Up the Executables

In this step, you will install four binaries into your system. The four binaries are:

crm - the CRM114 "engine", containing both the microcompiler and the runtime library. It's called "crm" because "crm114" is too hard to type.
cssutil - the .css file check/verify/edit program
cssdiff - the .css file diff program
cssmerge - the .css file merging program

One important point: do NOT install CRM114 or any of its utilities setup or said to root. If you do, that's just an invitation for someone to utterly hose your system without even trying. We're not talking an intentional attack, just an inadvertent command or script gone weird could do it.

There are three ways you can set up these executables. You can:

- install with a .rpm kit
- install with a .i386.tar.gz (tarball of statically linked binaries)
- install with a .src.tar.gz (tarball of complete source)

AGAIN: If you do not have root on the machine you are installing on, you may have some problems during the installation. You may want to reconsider using the statically linked binaries instead of compiling from sources.

Method A: Installing from .rpm

(note- we don't have a good RPM for the current rev, so this section is not really accurate)

Become root, then type:

```
        rpm -ivh crm114-<version>.rpm
```

and it'll all happen automagically. Now, you can test the install. A quick test is to type:

```
    crm -v
```

which should report back the version of CRM114 you have just installed. If this works, you can proceed on to the examples in this book.

Method B: Installing from .i386.tar.gz

This method takes a few more commands to perform. First, untar the binary release. Type:

```
      tar -zxvf crm114-<version>.i386.tar.gz
```

You should now become root. If you do not have root on your machine, you can execute CRM114 programs directly from your home directory, by changing your $PATH appropriately; see your shell man page for how to do this for your particular shell (it varies with the shell, so I can't tell you here how to do it) and skip to the end of this step.

Once you're root, type:

```
      cd crm114-<version>
      make install_binary_only
```

This will install the pre-built binaries of CRM1114 and the utilities into /usr/bin. This is the default install location for CRM114. If you want them installed in a different place, edit the Makefile and change INSTALL_DIR (near the top of the Makefile) to a different directory.

Note that if you type "make clean" you'll _delete_ your prebuilt binaries, so don't do that!

Now, you can test your work. Type

```
   crm -v
```

which will cause CRM114 to print out the version of itself you just installed. If this works, you can proceed with the examples in this book.

Method C:

This method is the most complex. Start by uncompressing and untarring the big .src.tar.gz with the command:

```
tar -zxvf crm114-<version>.src.tar.gz
```

Now `cd` down into the crm114-<version> directory. You will see many files here.

You now have a choice: you can build CRM114 with either the GNU regex libraries (not recommended, as GNU regex can't handle embedded NULL bytes and has other issues), or with the TRE regex library (recommended; this is what you get with the precompiled binary kit).

By default, you will use the TRE regex library; however, this means you have to build and install TRE before you build and install CRM114 itself. You can either grab the most recent version from the TRE homepage at http://laurikari.net/tre, OR you can use the version that is pre-packaged with your CRM114 download. (The pre-packaged version is tested against CRM114- the fresh one may have new features. Take your choice- it's good stuff either way)

Fortunately, building and installing TRE is easy. The TRE regex library can peacefully coexist on the same system as the GNU regex library.

To install TRE, become root, then type (**don't forget the** "**--enable-static**") :

```
    cd crm114-<version>
    cd tre-<tre_version_number>
    ./configure --enable-static
    make
    make install
```

You have now installed the TRE regex library as /usr/local/lib/libtre .

Depending on your system, you may need to also add `/usr/local/lib` to `/etc/ld.so.conf`, and then run `ldconfig` as root. Or not- some systems seem to run ldconfig automatically. If, during the next steps, you get annoying messages on the order of "can't find ltre" then `ldconfig` is the thing to try.

Once TRE is built and installed you can compile CRM114 and the accompanying utilities (cssutil, cssdiff, and cssmerge). By default, CRM114 installs into `/usr/bin` (_not_ `/usr/local/bin`) - if you want to change this, change the definition of INSTALL_DIR near the top of the file "Makefile").

Change directory back up to the CRM114 directory, then become root, then (noting that the current release doesn't use .configure, but future releases will) type:

```
make clean
make install
```

This will compile, link, install, and strip the executables (stripping gets rid of unnecessary debugging information and makes the executables load faster and use less memory).

You can test your installation of CRM114. Just type:

```
crm -v
```

and CRM114 will report back the version of the install. If this works, you can proceed with the examples in this book.

Testing your installation

If you *really* want to test your installation, you can run it against "megatest.sh", which attempts to test every code path in the system (well, all of the non-error paths at least). Coverage is incomplete, but at least it's a strong confidence indicator.

Note that this only works if you've installed the TRE engine. The GNU regex engine has enough "fascinating behaviors" that it will get a lot of things wrong; the GNU regex package also doesn't handle approximate regexes at all, and since those are in the test set, you'll error out on each of those as well.

The easy way to run megatest is:

```
make megatest
```

which will report back any differences between what your local install of CRM114 did and what the "known correct" results are. (Of course, these “known correct” results are really just what happens when a human looks through the pages of output and then rubber-stamps the result. There's no guarantee that any particular release of CRM114 is error-free; instead there IS a guarantee, that somewhere, somehow, the software will fail...)

If there are any differences between the supplied "megatest.log" and your own results, OTHER than process IDs in the "MINION PROC" results, please file a bug report and we'll figure out what went wrong (and remember to include the **`crm -v`** output, so we know exactly what version of the software you are running).

“But, for my own part, it was greek to me.”

– *Julius Caesar*, Act I Scene II
William Shakespeare

Appendix C: Minifilter.crm Program Listing

Here's the complete text of minifilter.crm, the mini email antispam filter explained in the text.

```
#!/usr/bin/crm114
{
      isolate (:conclusion:) /unknown/
      call /:triggerword_check:/
      call /:dispatch_result:/
      call /:classify_check:/
      call /:dispatch_result:/
      output / ERROR – this program should never get to here! \n/
      exit /99/
}

:dispatch_result:
{
      match [:conclusion:] /good/
      accept
      exit /0/
}
{
      match [:conclusion:] /spam/
      #     it's spam... find the subject header and change it.
      {
            match (:subj:) <nocase nomultiline> /^Subject:/
            alter (:subj:) /Subject: [[SPAM]]/
            accept
            exit /1/
      }
      #     what if there WAS no subject header?
      {
            alter (:_dw:) /Subject: [[SPAM]]\n:*:_dw:/
            accept
            exit /1/
      }
}
#     it's neither- just return
return
```

```
:classify_check:
{
      #      nest down one level to catch the fail...
      {
            classify <osb unique microgroom> ( good.css | spam.css )
            isolate (:conclusion:) /good/
            return
      }
      isolate (:conclusion:) /spam/
      return
}

:triggerword_check:
{
      input (:trigwords:) [triggerwords.txt]
      {
            #           Get the next trigger word (there's one per
line)
            match [:trigwords:] (:: :goodbad: :trig:) \
                  <nomultiline fromend> \
                   /(\+|-)(.*)/
            #
            #     Is that trigger word in the input text?
            {
                  match <nocase> /:*:trig:/
                  #              yes, :trig: is in the unknown text.
                  {
                        match [:goodbad:] /-/
                        #           it's bad
                        isolate (:conclusion:) /spam/
                        return
                  }
                  isolate (:conclusion:) /good/
                  return
            }
            #  no, this trigger word did not match.  Go try again.
            liaf
      }
      #  The “next triggerword” match finally failed.
      isolate (:conclusion:) /unknown/
      return
}
```

Index (draft)

Semi-edited – updates will be made, bug reports are greatly solicited.