Virtual Machine *CMS Pipelines pattern* **— A Filter for Pattern Matching Edition .33337**

March 4, 2009

John P. Hartmann

IBM Danmark A/S
Nymøllevej 91
DK-2800 Lyngby
Danmark

JOHN at EMEAVM1 (jphartmann@vnet.ibm.com)

Formatted March 4, 2009 11:45 a.m.

Acknowledgments

This filter owes a lot to SNOBOL4. See Griswold, Poage, Polonsky: *The SNOBOL4 Programming Language* (Second Edition), Prentice-Hall, 1971. Though *pattern* only matches, the way a pattern is matched is more or less the same, and the tutorial in the manual (using bead diagrams) is a useful introduction.

The first atoms written were based on the basic pattern variables in SNOBOL4.

The most enthusiastic user, Brent Longborough, kept me honest with the function during the phase when *pattern* grew to its current state. My Bookmaster Mentor, Kevin Minerley, kept me honest with the documentation.

Disclaimer

The program described herein (*pattern*) was written entirely for my own use and is not available as an IBM product.

This paper has not been submitted to formal review; the views presented are entirely my own.

Lyngby, June 1992. j.

Third Edition (June 1992)

This edition describes the function in edition .33337 of PIPSYSF MODULE. Changes since December 1989 are marked with a vertical bar in the margin. Most changes are editorial.

Table of Contents

Preface

What is *pattern*?

pattern is a filter for *CMS Pipelines* to perform complex context-sensitive searches and text replacements. For example, *pattern* can change all quotes to "&esq.", except for those in figures, GML tags, and examples.

pattern works by matching a pattern (its argument string) against each input record. Output records are built from input or literal data, or both; zero or more output records can be written while an input record is processed. Output records can be written to any stream. *pattern* has memory. You can store values in variables to be used when processing subsequent records.

Why *pattern*?

When you cannot formulate a task as a cascade of built-in filters, your alternatives are to write a REXX program to perform the function or to use *pattern*.

pattern is useful:

- When the function is so simple that you would rather not create a separate file with the REXX program. Write the pattern as the argument in the pipeline specification.
- When the function is so complex that REXX parse has to be combined with logic and things become cumbersome. Such a pattern is often stored in a separate file.
- When the REXX program runs too slowly. Though *pattern* is interpretive, it does not rely on CMS services directly; you can expect a speed-up of five times over equivalent REXX function.

Who is *pattern* for?

Thus, *pattern* may be useful for many people. It is particularly suited to the toolsmith, though its syntax is not like most other programming languages.

- SNOBOL old-timers are likely to find it easy to understand.
- E?GREPpers should find many concepts they already understand, though the syntax is more verbose. *pattern* can recognise strings from context-sensitive languages.
- REXX programmers yearning for more powerful parsing constructs should find *pattern* quite useful.

Publications

CMS Pipelines User's Guide, SC24-5609, describes how to use *CMS Pipelines*.

CMS Pipelines Reference, SC24-5592, describes the built-in programs and pipeline commands.

The guide and reference for the VMTOOLS version of *CMS Pipelines* is PIPUG LIST3820 (on VMTOOLS).

Overview

The pipeline filter, *pattern*, processes its argument string (the pattern) against each input record. Typically, these steps are performed:

1. Read a record from the primary input stream. The record is read with the locate-mode interface which blocks the left-hand neighbour stage while the record is processed.
2. Match the record against the argument string (which is called a pattern expression), loading data into an output buffer in the process.
3. Write the contents of the output buffer to the primary output stream, if the pattern matches the input record.
4. Reset the output buffer to be empty; the contents of the output buffer are discarded if the pattern does not match the input record.
5. Release the input record.

Thus, *pattern* is driven by input data; you do not program in the sense you write code for REXX. A pattern is more like the template in the REXX Parse instruction than it is like procedural code.

In general, the argument string to *pattern* is an expression, defined recursively:

1. An atom is a term of a pattern expression.
2. A list of terms is a pattern expression.
3. Two or more lists of terms separated by the keyword OR are a pattern expression.
4. Recursion: A pattern expression in parentheses is a term of a pattern expression.

The argument to an atom, if one is provided, is a string expression. This is very close to a REXX expression (but not precisely one). Many built-in functions are provided; most are equivalent to functions available in REXX.

Note: *pattern* implements concepts from SNOBOL4. If you do not know SNBOL, be forewarned that some of these concepts are subtly different from what you may expect if you are used to procedural languages.

Sample REXX Programs and Equivalent Patterns

Figure 1 on page 2 is a REXX program to copy its input stream to the output. The command environment is *CMS Pipelines*, rather than CMS; READTO and OUTPUT are pipeline commands. READTO loads the next input record into the variable; OUTPUT writes its argument to the output stream.

Figure 1. COPY REXX

```
/* Copy lines.  This releases a synchronous stage.                  */
/*                                John Hartmann 31 May 1989 13:07:28 */
signal on novalue
signal on error
do forever
   'readto in'
   'output' in
end
error: exit RC*(RC¬=12)
```

The COPY program is invoked by coding it between vertical bars in the pipeline:

```
pipe ...|copy|...
```

The equivalent pattern is trivial; it instructs *pattern* to copy the input record to the output:

```
pipe ...|pattern rest:output|...
```

This example was rather unkind to REXX; Figure 2 is the reverse; it shows a sample REXX program which reads input records, parses them, and writes output records.

Figure 2. Sample REXX Program

```
/* Sample REXX parsing program                                      */
/*                                John Hartmann 27 Apr 1989 18:17:16 */
signal on novalue
do forever
   'readto in'
   If RC/=0                            /* EOF?                      */
      Then exit                        /* Yup, get out              */
   parse var in del+1 string (del) rest
   'output' 'String length' length(string) 'value' rest'.'
end
```

The equivalent pattern stage is shown in the context of a simple pipeline in Figure 3. The figure shows what you write in the pipeline specification; no additional files are required.

Figure 3. A Sample Equivalent Pattern

```
/* Pattern sample EXEC */
'pipe < input file|',
   'pattern arb:del to(del):string arb',
       'output("String length " length(string) " value ")',
       'rest:out output(".")|',
   'console'
```

The pattern expression is the argument string to pattern. The whole pipeline is written as a REXX expression continued over three lines. Figure 4 on page 3 describes each atom in the pattern expression.

Figure 4. Pattern Explained	
`arb:del`	Scan one character (the first on a line) and assign its value to the variable `del`.
`to(del):string`	Scan up to the next occurrence of the character and assign the value scanned to the variable “string”.
`arb`	Discard the second occurrence of the character.
`output(...)`	The second line writes the result of the expression into the output buffer. Unlike REXX, a blank in an expression means catenate without a blank; thus, the blanks around the string are included in the literals.
`rest:out`	Append the remainder of the input record, if any, to the contents of the output buffer.
`output(".")`	Append a period to the contents of the output buffer.

The examples Figure 2 on page 2 and Figure 3 on page 2 give the same result when an input line contains a properly delimited string, but REXX and *pattern* differ markedly when the string is not properly delimited: REXX assigns null values to a variable for which there are no data; *pattern* fails instead. The example in Figure 3 on page 2 discards records that are not properly delimited. A pattern expression can have alternatives to perform when matching fails; in contrast, “if” instructions must be used in REXX to determine if input data have the correct format.

pattern Concepts

Cursor

The cursor points into the input record; it moves from left to right as the pattern expression is matched. There is also a pointer into the pattern expression to the "thing" being matched at any one time; this pointer has no special name.

The cursor sits between characters. It starts out before the first character of the input record; this is position 0.

Left Margin

The cursor position before the first character of the current input record. The cursor is put in the left margin when a record is read.

Right Margin

The cursor position after the last character of the current input record. When a null record (one of zero length) is read, the cursor is both in the left and the right margin.

String Expression

The argument to an atom is an expression similar to a REXX expression. The differences are noted in "String Expressions" on page 42; the major ones are:

- Numbers are integers; fractions and exponents are not supported.
- Unset variables are null: a reference to a variable that has not been set returns a string of length zero.
- A null string converts to the number zero when used with an arithmetic operator. (But the integer zero converts to the string of one character with a single zero.)
- The relational arigthmetic operators (=, >, <, and their variations) treat a null string or a string of blanks as zero by nature of the automatic conversion to zero.
- The assignment operator (:=) assigns a value to a variable. The result of the assignment operator is the value set, that is, the right-hand side of the operator; an expression can have multiple assignment operators.
- The discard operator (;) evaluates two expressions. The result of the left-hand expression is the result of the discard operator. The result of evaluating the right-hand side is discarded; it is evaluated only for side effects (often assignment).
- Variable names are folded to uppercase.

Incompatibilities with REXX expressions

- The blank operator catenates two strings without inserting a blank. Write a string with a blank when you wish one.
- The blank operator has a different precedence than in REXX; use parentheses when expressions are assigned to variables.

Variables

Variables are set by

- the assignment operator in a string expression, and
- pattern matching.

Variables set by the assignment operator are permanent; they retain their value until reset by assignment or pattern matching. If a variable that was set by assignment is later set by pattern matching, it reverts to being ephemeral.

Variables set by pattern matching contain a substring of the input record. Such variables are discarded with the input record when the next record is read.

Variable names

- Variable names begin with an underscore, a letter, or one of the national characters shown in Figure 39 on page 53. Letters are a-z and A-Z. National characters are a subset of the code points defined for national use on 3270.
- Names with more than one character or underscore have the additional characters made up from letters, underscores, national characters, and numbers.
- Variable names cannot be longer than 250 characters.
- Variable names are translated to uppercase using the translate table pointed to by the contents of the field NUCUPPER in NUCON. As shipped with CMS this translates a through z to A through Z. It does not change the national use code points. On MVS uppercasing is performed using a table translating the English letters only; the MVS table is hardwired into *CMS Pipelines*; it cannot be modified.

Figure 5. Variable Names

```
a    Z    {bler    b}de     #7    $6!!
```

Figure 6. Not Variable Names

`q.fred`	The period is not allowed in a variable name.
`2u`	The first character must be a letter or a national character.

The contents of variables that have not been assigned a value are considered to be the null string.

At the moment, compound variables (for instance, q.fred) are not supported; however, the name of the variable to assign a value can be computed as shown in Figure 7 on page 6. Use the built-in function VALUE to refer to the value of a

variable whose name is computed. Computed names are truncated after 250 characters and folded to uppercase; any character is valid in a computed name.

Figure 7. Using a Computed Variable Name in an Assignment

```
('q.' fred):='the value'
```

The Output Buffer

Records written by *pattern* are built in an area called the output buffer. Data loaded in the output buffer are appended to the previous contents of the buffer. You can append data from

- the input record,
- expressions you compute, or
- a combination of these.

Flushing the Output Buffer

You can flush the output buffer before the pattern is completely matched; this writes the contents of the buffer so far to an output stream and resets the buffer to be empty. You can select which stream to write the buffer by an optional argument; the default is to write to the primary output stream.

When the pattern is matched and the output buffer has one or more characters, *pattern* automatically writes this to the primary output; you need not use an explicit flush for this, unless you wish to write a null record.

Atom

An atom is the smallest unit of matching. Atoms are listed in alphabetical order in “Atom Reference” on page 26.

An atom is an identifier and optionally an argument string in parentheses. Figure 8 shows an example of an atom.

Figure 8. A Sample Atom

```
string("abc")
```

This looks like a REXX function call, but unlike REXX, the atom does not return a result; it either matches or fails.

Match and Fail

In the example in Figure 8, matching the atom is done by testing if the string is present after the cursor position. If the string is present, the atom matches, and the cursor is moved beyond the string it matches.

The atom fails when, in this particular case, there are fewer than three positions between the cursor and the right margin, or if the string does not match the contents of the input record from the cursor position onwards.

In general, an atom fails when a condition is not satisfied; the cursor moves from left to right when the atom matches (though it can move zero positions).

Types of Atoms

It is convenient to group atoms into three categories:

- Atoms that match and move the cursor, or fail.
- Atoms that either match or fail, but do not move the cursor.
- Atoms that never fail and do not move the cursor.

Atoms Moving the Cursor

When a “real” atom is matched, the cursor moves to the right (maybe zero places, but to the right all the same). Such an atom fails, for instance, when data at the cursor are not a specified string.

The substring of the input record matched by the atom is the string between the cursor position before and after matching the atom; it can be assigned to a variable, and appended to the output buffer.

Some examples of this kind of atoms are shown in Figure 9.

Figure 9. Some Real Atoms	
`any("abc")`	Matches a single character which is either a, b, or c.
`"abc"`	Matches the string of three characters; the atom STRING is assumed for a literal string or a variable.
`tab(5)`	Moves the cursor between the fifth and sixth character of the input record, assuming the cursor is at or to the left of this position. It fails when the cursor is to the right of the fifth position.
`rem`	Match to the end of the record. REM cannot fail; it matches the null string in the right margin.

Conditional Atoms

These are also called predicates. They are atoms that fail or match depending on some condition, but do not move the cursor. In jargon, they “match the null string” if the condition is satisfied.

Figure 10. Some Predicate Atoms	
`pos(5)`	Fails unless the cursor is between the fifth and sixth character in the input record.
`rpos(0)`	Test if the cursor is in the right margin. It fails when the cursor is not in the right margin. This is like POS, but counting from the right of the record towards the left.

Atoms that Never Fail

Atoms that do not move the cursor and never fail are matched to obtain side effects, and to complete an expression.

Since the cursor does not move, these atoms also match the null string.

Figure 11. Some Never-fail Atoms	
`output("abc")`	Append the string to the contents of the output buffer.
`flush`	Write the current contents of the output buffer to the primary output stream and reset the buffer to be empty.
`null(notfirst:=1)`	Evaluate the expression to set the variable to 1.

Assignment by Pattern Matching

Put a colon and a variable name after an atom to assign the string matched by the atom to a variable. The variable is reset to the null string when the next input record is read.

When you assign a value to one of the simple variables OUT, OUTP, OUTPU, and OUTPUT, the string is also appended to the output buffer.

Figure 12. Assigning Values to Variables	
`len(5):out`	This assigns the next five characters from the cursor position to the variable “out” and also appends this string to the output buffer. This is the most efficient way to append to the output buffer. It is also the shortest one to type.
`tab(7):var` `output("it is: " var)`	This example shows a sequence of two atoms. The first one moves the cursor after the seventh character of the input record (assuming it is not already beyond this position) and assigns the string to the variable “var”. The second atom appends a literal and the contents of the variable to the output buffer.

Use assignment in a string expression when you wish to retain the value of a variable after the next record is read.

Figure 13. Assigning Permanent Values to Variables	
`word:out` `null(stored:=out)`	The first atom assigns a blank-delimited word (or the remainder of the record) to the variable “out”, and appends it to the output buffer. The second atom does nothing as far as pattern matching is concerned; use it to write a string expression to assign the value to the permanent variable “stored”.
`word:var` `null(var:=var)`	In this example, the word is stored in a permanent variable without being written to the output. The string expression may look redundant, but it makes the variable “var” persistent.

pattern Expressions

Subsequentation

A very long and very fine word for doing things in sequence. A list of atoms matches when all atoms in the list match; they are tested from left to right in the order written.

Figure 14. Sample of a List of Atoms

```
any("abc") span("0123456789")
```

The example in Figure 14 shows a list of two atoms. It matches one of the letters a, b, or c followed by one or more digits.

Alternatives

Use the keyword OR to separate alternative lists of atoms. There can be more than two alternatives, so this can also be thought of as a “case” construct.

One of the alternatives must match for the pattern as a whole to match. Alternatives are tested in the order written. When testing an alternative, the cursor is moved back to where it was when the expression was “entered”, to test the first alternative. Side effects, for instance assignments and output, are **not** undone.

Figure 15. Sample Alternatives

```
any("abc") span("0123456789")
   or
to(":") word(". ")
```

Figure 15 shows an expression with two alternatives each of which consists of two atoms. Two types of strings are matched:

1. Strings beginning with one of the letters a, b, or c, followed by one or more numbers.
2. Strings having a colon in them up to the next blank or period. This (naively) looks for a GML tag.

Grouping with Parentheses

Parentheses group atoms. OR has lower precedence than subsequentation; use parentheses when you wish to have alternatives in a list of atoms.

Figure 16. Sample Grouping	
`"a" len(5) or rest`	The first list matches string beginning with “a” which are at least 6 characters; the alternative matches anything. Thus, this expression never fails.
`"a" (len(5) or rest)`	The first atom matches the character; the expression in parentheses matches the next five characters or the remainder of the record, if less than five characters remain in the record. The complete expression fails if the first character is not “a”.

Term

A term is an atom or an expression in parentheses. The substring of the input record matched by a term is assigned to a variable just like the string matched by an atom.

Nesting Pattern Expressions

The amount of virtual storage available is the only limitation to the nesting of *pattern* expressions.

Iteration

A term (an atom or a pattern expression in parentheses) is iterated when it is matched repetitively.

Special characters after a term indicate how it should be iterated:

? Zero or one time.
* Zero to infinity.
+ One to infinity.
= A specific number of times. Write a range as a minimum and a maximum number separated by a hyphen.

Note: The iteration character is before variable assignment, if any.

First time an atom is met, *pattern* matches it the minimum number of times. The minimum number of times (sometimes zero) is matched unless a failure of a subsequent atom forces further iteration. This may be counter-intuitive; see the example in “NULL [(String)]” on page 35.

When an assignment is made as a result of matching an iterated term, the value assigned begins with the position of the cursor before the first time the term was iterated. For example, in the expression:

```
arb*:var
```

the first time ARB is matched the variable is set to one character, the next time to two characters, and so on. However, in the expression:

```
(arb:var)*
```

the assignment is made as the result of matching an atom which is not iterated, so the variable is set to one character each time ARB is matched.

Warning: Though you probably think of iteration in the way of procedural programming, this breed is different. The iteration character specifies that the atom can be iterated; it is iterated the *minimum* number of times when pattern matching is moving forward. Any further iteration is caused by failure of a subsequent atom or expression. It is often the failure of a subsequent predicate that drives the iteration.

What Happens when an Atom Fails?

The cursor does not advance when an atom fails. *pattern* goes back through matched expressions to find an expression that it can iterate. *pattern* tries the alternative when nothing is found to iterate.

- Iteration is possible:

  If a term (an atom or an expression) can be iterated further, the cursor is moved back to the position after the last iteration (possibly going back over strings matched by atoms since then), and the expression is matched one more time.

- Iteration is not possible:

  When no iteration can be done, the cursor is moved back to where it was at the beginning of the pattern expression, and the next alternative is matched.

- No iteration is possible and no alternative matches:

  If no expressions can be iterated and no alternatives are matched, the expression fails, and the failure propagates to the containing expression. This failure is processed just as when an atom fails.

The Fence

Put a fence when you do not wish a failure to go back or look for alternatives at a particular level of expression. Use FENCE to improve performance when you know that there is no point in trying alternatives. You can also use it when you do not wish the alternative to be tried as described in “If-Then-Else” on page 15.

Distributing Assignment

These two expressions perform the same function:

```
to("abc"):out rem:out
(to("abc") rem):out
```

This, however, is a special case because REM cannot fail. The following two expressions do not always give the same result:

```
after("abc"):out to("def"):out or null
(after("abc") to("def")):out or null
```

The first expression produces output for all lines with `abc` in them; the string up to `def` is also included if it is present after `abc`. Lines without `def` after `abc` cause a failure in TO, but the contents of the output buffer loaded by the first assignment

are retained. In the second expression, the assignment is performed when both atoms are matched; no assignment is done when either of the atoms fails.

Pattern File

Write the pattern expression as the argument to *pattern* or put it in a file and include it using the facility described in “%INCLUDE Facility” on page 21. You can nest includes. Pattern files can be loaded in storage with EXECLOAD; this improves performance when a pattern file is used over and over again. This uses the pattern in the file TEST PATTERN:

Figure 17. Using %include

```
...|pattern %include(test)|...
```

Though simple patterns are easy enough to write as arguments on the filter, complex ones with many strings are easier to write if you use a pattern file where you do not have to worry about putting REXX quotes around constants, along with all the quotes in the pattern itself.

Writing a Pattern Expression

Case is respected in strings.

Case is ignored (folded using the NUCUPPER translate table) in:

- Names of atoms,
- Names of variables (but this may change), and
- Built-in functions in string expressions.

White space is required to delimit an atom from another one, and the keyword OR. White space consists of:

- Blanks.
- End-of-line in an embedded file.
- Comments. Comments are REXX-style; you can nest them.
- %include orders; you can nest includes.

There must be no white space before the left parenthesis of an atom or a built-in function reference; a left parenthesis that follows white space groups an expression.

Figure 18. Correct Pattern Expressions

Pattern	What it Does
`word:label toword word:opcode toword rest:operands`	Scan the label and operation code of an Assembler statement. The operands and comments, if any, are scanned into the variable `operands`.
`("abc" or any("def"))* rpos(0)`	Ensure that the record is null or has one or more occurrences (in any order) of the string abc or characters d, e, or f.
`arb* "magic"`	Ensure that the input record has the string “magic” in it somewhere, for instance in “unmagical”.
`to("magic")`	The same as the previous example using an atom instead of iterating an expression. This is much more efficient. Always try to write goal-directed patterns.

Figure 19. Incorrect Pattern Expressions

Bad Sample	What Is Wrong with it
`any`	Parentheses with a string expression are missing; ANY must have an argument.
`/* A /* comment */`	Comments are not properly nested; the first comment is not closed.
`( string("abc")`	Parentheses are not balanced.

Sample Pattern that Replaces Blanks with an Asterisk

Figure 20 shows a complete pattern and an explanation of its constituents.

Figure 20 (Page 1 of 2). Replace Run of Blanks with an Asterisk

`(break:out span out("*") or rest:out)* rpos(0)`

Pattern Segment	What the Pattern Segments Does
`(break:out span out("*") or rem:out)*`	A pattern expression in parentheses followed by an asterisk to indicate iteration from zero to infinity. First time this is matched, it matches the null string.
`rpos(0)`	Test for the cursor being in the right margin (zero to the left of the right margin). If the input record is null, the pattern expression is now matched. When the input record is a character or more, RPOS fails first time and the expression in parentheses is matched once more. The cursor is still in the left margin.
`break`	Locate the next blank. The cursor moves to be in front of the first blank in the record. BREAK fails when there are no blanks to the right of the cursor.
`:out`	Set a variable named “out” to the matched string. The matched string includes everything up to the blank. A variable with a name that is an abbreviation of “output” gets its contents written to the output buffer.
`span`	Skips or “spans” blanks until it finds a non-blank or hits the right margin. There must be at least one blank, but we know this is the case because BREAK matched.
`out("*")`	Writes the character (or literal string) “*” to the output buffer.
`or`	Keyword to delimit alternatives. The following atom is matched if BREAK fails.
`rem:out`	Matches any characters up to the end of the record ensuring the last word (if any) gets written to the output buffer

Figure 20 (Page 2 of 2). Replace Run of Blanks with an Asterisk	
`rpos(0)`	Test for the cursor in the right margin. When the cursor is not in the right margin, RPOS fails and the previous expression is iterated. This mechanism ensures that the complete record is processed. When the input record is null, the cursor is in the right margin after no iterations and the record is completely processed. *pattern* discards a null output record unless you force it out with FLUSH. When there is one word in the input line, `rpos(0)` matches after one iteration; two iterations are required to process two words, and so on. Note that it is the failure of `rpos(0)` that drives the iteration; without it any input record would be matched immediately and no output produced.

The built-in function SPACE can do a similar thing while stripping leading and trailing blanks. The example in Figure 20 on page 14 produces a leading and/or trailing star if the input record has leading and/or trailing blanks.

```
pattern output(space(input(),,'*'))
```

INPUT is a built-in function that returns the complete input record.

If-Then-Else

Write if-then-else constructs with the atom IF in this manner:

Figure 21. If/then/else
`(if(<expression>) fence <then-clause> or <else-clause>)`

Note that <then> and <else> clauses are pattern expressions, not string expressions. (Use NULL to evaluate an expression for its side effects.) The atom FENCE prevents failure in the then-clause from trying the else-clause as an alternative; it is not required if the then-clause cannot fail.

Multiple tests are possible with the atom IF.

```
(
if(<expression-1>)
   fence <then-clause>
or
if(<expression-2>)
   fence <then-clause>
or
   <otherwise>
)
```

A test can often be done without using the atom IF, as shown in these two equivalent ways of incrementing variable `v` if `a` is positive.

```
(if(a>0) null(v:=v+1) or null)
null(v:=v+(a>0))
```

The Output Buffer

Output records are built in the output buffer. Its contents are discarded when a record is read.

Getting Data into the Output Buffer

Data are loaded into the output buffer:

- As a side effect of assigning matched input data to variables that are an abbreviation of OUTput, down to three characters.
- By the OUTPUT atom.
- By the OUTPUT built-in function.

Data loaded in the output buffer stay there even when the expression loading the string later fails and the cursor is backed up before the string. When the result of an iteration is loaded in the output buffer, a string is appended after each iteration.

Getting Data to the PIPE from *pattern*—Using FLUSH

The contents of the output buffer are written to the pipeline when:

- The atom FLUSH is matched.

  If the output stream is not connected when writing with the atom FLUSH, the argument to the atom EOF is evaluated for side effects, and processing stops with return code 0.

- The FLUSH built-in function is called

  When the FLUSH built-in function is used to empty the buffer, the result is a null string if the buffer was written OK; the string "1" is returned when the output stream is not connected.

- After the pattern matches an input record.

  Processing stops with return code 0 without evaluating the argument to EOF when a pattern is matched and the primary output stream is not connected.

The argument, if present, to the atom or function FLUSH, designates the stream to receive the record; it defaults to the primary output stream.

There is one output buffer for all streams. Store data in variables when building multiple records in parallel.

It is an error to select an undefined stream with FLUSH or the built-in function. Processing terminates with an error message.

Examples of Getting Data into the Output Buffer

Figure 22. Iterating Assignment to OUTPUT

```
pipe literal abc|pattern arb*:out rpos(0)|console
aababc
R;
```

Figure 22 is a complete pipeline with the CMS response. It shows how iteration interacts with assignment to the output variable. Figure 23 shows the components of the pattern, with a description.

Figure 23. Loading Data in the Output Buffer

Pattern Segment	What the Pattern Segment Does
`arb*:out`	The first time, zero iterations are done and the atom is not matched. This means that a null (empty) string is matched and appended to the output buffer. This leaves it empty.
`rpos(0)`	The cursor is not in the right margin, so RPOS fails. *pattern* backs up to the previous atom that can be iterated.
`arb*:out`	ARB is matched. It matches any character and moves the cursor to be between the a and the b, loading the first a in the output buffer. This process continues. On the next iteration, “ab” is appended to the output buffer, and so on.

Figure 24. Iterating Assignment to OUTPUT

```
pipe literal abc|pattern arb*:out flush rpos(0)|console

a
ab
abc
R;
```

Figure 24 shows the result when FLUSH is matched on each iteration.

The result is that a record is written after each iteration; the blank line represents the initial null record.

Figure 25. Iterating assignment to OUTPUT

```
pipe literal abc|pattern (arb* rpos(0)):out|console
abc
R;
```

Figure 25 shows the result when only the complete result of the expression is assigned to the output variable.

Note: REST performs the same function as the expression and is much faster.

String Expressions

Arguments to atoms are called string expressions. They compute a number or a string of characters, as required by the atom.

String expressions are like REXX expressions with a few differences noted above; the most important ones are the assignment *operator* and the separator operator.

A sequence of digits is a numeric string. It is not converted to integer before it is determined that an integer is required. Thus ANY(089) does see the string of three characters, whereas ANY(089+0) sees a string of 89 because the leading zero is discarded in the conversion to integer when the expression is evaluated.

Assignment Operators in String Expressions

To set a simple variable, put its name before the colon-equal operator. The example below shows how to maintain a record count.

Figure 26. A sample Assignment

`null(recno := recno+1)`	
`:=`	The colon-equal operator assigns in the way “we are used to”.
`recno :=`	The variable name is at the left-hand side of the operator.
`:= recno + 1`	The expression is on the right of the operator. The variable “recno” must be the null string, or numeric.
`:=`	Remember the colon.

You can assign an expression to several variables; you can use the result of the assignment operator as a term in a larger expression.

Figure 27 (Page 1 of 2). Multiple Assignments in an Expression

String Expression	**What it Does**
`a:=b:='string'`	The literal `string` is assigned to two variables, `a` and `b`.
`c:=1+b:=1+a:=0`	The variable `a` is set to zero. One is added to this result and assigned to `b` which becomes one. `c` is assigned the value 2.
`'$' (n:=n+1)`	The variable `n` is incremented and the result is appended to the literal dollar sign.
`var := ('$' n:=n+1)`	The variable `n` is incremented and the result is appended to the literal dollar sign. The result is assigned to the variable `var`.

Figure 27 (Page 2 of 2). Multiple Assignments in an Expression	
String Expression	**What it Does**
`agg:=(var := '$' n:=n+1)`	The variable `n` is incremented. The variable `var` is assigned the literal dollar sign. The result is the two variables catenated; this is assigned to `agg`.

Figure 28. Not an Assignment	
`null(recno = recno+1)`	The equal sign is the operator that tests for equality of two terms. This atom evaluates the result 0 which is discarded by NULL; no assignment takes place.

The Separator

Use a semicolon in a string expression to separate assignment operators. The semicolon operator discards the result of the expression on the right-hand side; it is evaluated only for its side effects.

The result of the semicolon operator is the expression to the left of it.

Figure 29. Using Separator Operator	
String Expression	**What it Does**
`a:=1; b:=2`	Two assignments are performed. The result of the expression is 1.

Computed Variables

The name of a variable to assign can itself be computed in an expression. Generated variable names are put in an expression in parentheses on the left-hand side of the colon-equal operator. Parentheses with a string or a single variable reference is taken to be a simple name; write an expression, for instance prefixing a null string (`''var`), to assign to the variable whose name is the contents of VAR.

Write parentheses with an expression after the colon when you wish to compute the name of the variable to assign the result of pattern matching.

Input File:	**Pattern:**	**Output File:**

`abc`	`null(fred:='')`
	`rest:('q.' fred)`
	`output('Fred ="' fred '"')`
	`flush`
	`output(value('q.' fred))`
		`Fred =""`
		`abc`

Example of Computed Variables

Figure 30 loads the variables $1, $2, ... with the words of the input record and sets N to the number of tokens met.

Figure 30. TOKENISE PATTERN: Assignment to a Computed Variable

```
/* Tokenise the input record.  Assign variables &n                    */

null(n:=0)                              /* Reset counter              */

/*********************************************************************/
/* Skip leading blanks                                               */
/*********************************************************************/
(span(" ") or null)

/*********************************************************************/
/* Iterate over the words on the input line                          */
/*********************************************************************/

(
   word:("$"(n:=n+1))                   /* Take a word                */
   (span(" ") or null)                  /* Skip to next word          */
)*                                      /* Iterate                    */

rpos(0)                                 /* Until the end              */
```

Assuming the file in Figure 30 is called TOKENISE PATTERN, then this is a way to list the number of words on input lines.

```
...|pattern %include(tokenise) output(n " words on the line")|...
```

Doing Things Before and After the File Is Processed

There are two special atoms, TOF and EOF. They must be first in the pattern; they can occur at most once each. The argument is a string expression; it is evaluated for side effects; the result is discarded. If the output buffer is not empty after the expression has been evaluated, the contents of the output buffer are written to the primary output stream and the buffer is reset to be empty.

The string expression in TOF is evaluated before the first record is read. You can initialise variables using this atom. The difference between TOF and ONCE is that the argument to TOF is evaluated also if there is no input; ONCE is not matched before at least one record has been read.

The argument to EOF is evaluated after end-of-file is received; this atom is useful to write the contents of a buffer that is stored in a variable. Figure 31 shows how JOIN * is written with *pattern*. The contents of the variable “buffer” is by default null, so it is not explicitly assigned a null value at the beginning.

*Figure 31. JOIN * as a pattern*

```
...|pattern eof(output(buffer)) rem:r null(buffer:=(buffer r))|...
```

The NULL atom is matched for each input record; the record is appended to the buffer variable; no output is written while input records are read.

%INCLUDE Facility

This facility allows you to include pattern definitions from files stored on disk so you do not have to worry about escaping and nested quotes, as you must when the pattern is coded as the argument to *pattern*.

You can nest `%include`s to any practical depth. In 1988 the limit was 1365; it may change over time, but it is expected to remain a very large number. The CMS file system would need at least 6M of buffers to have so many files open concurrently. There are no restrictions on the number of include files that can be referenced in a pattern.

You can write include wherever you can put a blank or a comment. The case of the instruction is immaterial; on CMS the argument is used to invoke < which processes a mixed-case argument in its own way.

On MVS, *qsam* is used to read a member from the partitioned dataset. The first word of the argument specifies the member name. The second word (if present) specifies the DDNAME; the default is PATTERN.

Figure 32. %include Syntax

```
INCLUDE(<fname> [{<ftype>|PATTERN} [<fmode>|*]])  /*  CMS  */
INCLUDE(<member> [<DDNAME>|PATTERN]               /*  MVS  */
```

Notes:

1. A nested `%include` instruction must be complete on a single line of the file.
2. The `%include` instruction cannot have comments between the parentheses specifying the file identifier.
3. Literal strings cannot span records in an imbedded file.
4. Pattern files can be EXECLOADed under CMS. To reach such files, you must not specify a file mode; write the file name and optionally the type of the file. Write the file mode or an asterisk when you wish to be sure you are using the version on disk.

Summary of *pattern* Processing

pattern does the job in several stages:

1. Parse the argument string. *pattern* parses the argument string on commit level -1.

   A tree representation of the pattern and string expressions is built. Any errors in this processing causes *pattern* to terminate without reading any input records.

2. Commit to level 0. Quit if the return code is non-zero indicating failure of some other stage.
3. If a TOF atom is present, the expression is evaluated for its side effects; the result is discarded. This happens before the first record is read.
4. Process the input file.

pattern does not delay the record unless the atom FLUSH or the built-in function is used. Each input record is matched against the pattern proper (ignoring TOF and EOF).

5. After end-of-file is received the expression in the EOF atom, if present, is evaluated for its side effects; the result is discarded. This is normally to flush a stored value.

Tracing Matching Activity

Activity is traced to the stream with identifier “trc”, if defined for the stage. When defined for the primary stream, trace data are intermixed with normal output.

```
...|x.trc:pattern ....
```

If a stream is defined with identifier “xtrc”, the control block structure built, and expression evaluation are also traced; this is intended for debugging, but it may be used as a last resort when you get unexpected results.

Tricks of the Match-making Trade

Re-scanning the Input Line

When keywords can be in any order on the input line, you can scan to a keyword, find out what it is, and then process it accordingly. Or you can scan the line for a particular keyword, irrespective of its placement, as the example with FAIL shows.

TOKENISE PATTERN shown in Figure 30 on page 20 leaves the cursor in the right margin. Figure 33 shows how to process the input record word by word after the number of words is determined by the included pattern. The REXX program PI REXX is short-hand to include a pattern file; in this example the stage is `|pattern %include(anotl)|`.

Figure 33. ANOTL PATTERN: Reprocess the Input Line

```
/* Annotate input file                                              */

%include(tokenise)   /* Tokenise the input line; set number of words */

fail or                            /* Force a failure to restart scan */

nextword           /* Skip leading blanks; don't fail on null record */
null(line:=line+1; wc:=0)             /* Initialise counters         */
(
   output("Line " line " word " (wc:=wc+1) " of " n ": ")
   word:out                   /* Scan a word onto the end of the line */
   flush                              /* Write output line           */
   nextword                           /* Skip blanks                 */
)* rpos(0)                            /* Until all is done           */
```

```
 pipe literal abc def|pi anotl|console
►Line 1 word 1 of 2: abc
►Line 1 word 2 of 2: def
►Ready;
```

pattern Reference

Each atom is described in a section beginning with a heading in the left margin and ending with an example showing input, pattern, and output.

Abbreviations

In the heading, the minimum abbreviation is shown in uppercase; any further characters may be written with the same result.

Parameters

Some atoms do not require parameters; simply write the name. Such atoms, for instance ABORT, are shown with a single word in the heading.

Some atoms have an optional parameter; the type is indicated in square brackets after the atom name. BREAK is an example of such an atom.

Atoms that require a parameter, for instance ANY, have the type in parentheses after the atom name.

The name of the STRING atom may be elided when matching a single literal string or the contents of a single variable. Use an explicit STRING when an expression must be evaluated or when referencing a variable whose name is also the name of an atom.

Types of Parameters

The argument required for an atom is shown in parentheses after the name of the atom. There are three types of parameters:

Integer The atom requires a numeric argument that must be zero or positive. The result of the expression must be a string that can be converted to a positive number or zero; processing stops at runtime with a message if the conversion fails.

String The atom requires a character string. An expression is evaluated and converted to a character string. The order of the characters in the string is significant.

Enumerated The atom requires a list of characters. The argument is evaluated and converted to a string of characters. The order of the characters is not important; a character can be duplicated without changing the result.

The Most Useful Atoms

Check out these atoms first: CASEI, LEN, NEXTWORD, REST, STRING, TO, TOWORD, and WORD.

Atom Reference

ABBREV (String)

Never fails. Match leading characters from the argument string. ABBREV matches the null string in the right margin, when the argument is null, and when the character after the cursor is not equal to the first charecter in the argument string.

ABBREV is similar to the REXX built-in function, but not the same.

Figure 34. REXX and pattern ABBREV

```
parse var rest word rest
if abbrev('user', word, 1) then ...
```

```
'u'                                  /* minimum abbreviation       */
abbrev('ser')                        /* What is optional           */
(' ' or rpos(0))          /* Ensure it ends in a blank or the margin */
```

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	abbrev("Serenade"):out output("*")	* * Se* *

ABBREVCI (String)

Never fails. Match leading characters from the argument string, regardless of case. ABBREV matches the null string in the right margin, when the argument is null, and when the character after the cursor is not equal to the first charecter in the argument string, regardless of case.

ABBREVCI is similar to the REXX built-in function, but not the same.

Figure 35. REXX and pattern ABBREVCI

```
parse upper var rest word rest
if abbrev('USER', word, 1) then ...
```

```
casei('u')                           /* minimum abbreviation       */
abbrevci('ser')                      /* What is optional           */
(' ' or rpos(0))          /* Ensure it ends in a blank or the margin */
```

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	abbrevci("Serenade"):out output("*")	* s* Se* *

ABORT

Always fails. There is no back-up; the contents of the output buffer are discarded. The next input record is read and processed.

ABORT is useful when you wish to discard records that match a list of atoms or expressions.

The example looks for lines beginning with a colon and having the string “file” in them somewhere. These lines are discarded; other lines are copied to the output.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	":" to("file") abort or rest:out	sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.

AFTER (String)

Position the cursor after the next occurrence of the argument string. AFTER fails when the argument has more characters than there remains in the record, and when the string does not occur to the right of the cursor. These two expressions are equivalent.

```
after(string)
(to(string) len(length(string)))
```

See also AFTERCI, TO, and TOCASEI.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	after("fi") rem:out	le is a d=samp. for

AFTERCI (String)

Position the cursor after the next occurrence of the argument string, ignoring case. String and data bytes are translated to uppercase before being compared. AFTERCI fails when the argument has more characters than there remains in the record, and when the string does not occur to the right of the cursor. These two expressions are equivalent.

```
afterci(string)
(tocasei(string) len(length(string)))
```

See also AFTER, TO, and TOCASEI.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	afterci("fi") rem:out	le is a gref refid=samp. for

ANY (Enumerated)

A single character is matched if it occurs in the argument string. The converse is NOTANY.

The example selects the first character from records beginning with an s in either uppercase or lowercase.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	any("Saints"):output	s S

ARB

Any single character is matched. Though infinitesimally faster, ARB is equivalent to LEN(1). ARB fails when the cursor is in the right margin. ARB is useful after BREAK to skip the delimiter character.

SNOBOL4 gurus should note that arb* behaves like the ARB they know.

arb* is used to indicate that any number of “noise” characters are to be skipped. This usage is not the most efficient; often you can use one of the streaming atoms to get to the next occurrence of, for instance, a string.

This example deletes the beginning of the record to the character after the first colon.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	after(":") arb rem:out	.This file is a p1.pattern:ehp1.. igref refid=samp. for

AT (String)

The null string is matched if the characters from the cursor match the string.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	at("s") rem:out	sample for :hp1.pattern:ehp1..

ATCASEI (String)

The null string is matched if the characters from the cursor match the string irrespective of case.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	atcasei("s") rem:out	sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for

ATANY (String)

The null string is matched if the character after the cursor matches one of the characters in the string.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	atany("master plumber") rem:out	sample for :hp1.pattern:ehp1.. more information.

BAL

Match characters to the end of the record or to the first unbalanced right parenthesis, whichever occurs first. The concluding parenthesis is not included in the matched string. BAL fails when the remaining input has more left parentheses than right parentheses in a way where they are not balanced at any time.

The opening parenthesis should have been scanned before BAL is matched.

Input File:	Pattern:	Output File:
abcdef	bal:out	abcdef
abc(def)ghi	bal:out	abc(def)ghi
(abc(def)ghi)rest	'(' bal:out	abc(def)ghi
(abc(def)ghi	bal:out	*fails*

BREAK [(Enumerated)]

Characters are matched up to (but not including) the first occurrence of a character in the argument string. With no argument, characters are matched up to the first blank. The string matched can be null. Break fails in the right margin or if none of the argument characters are present in the remainder of the record.

See also TO, TOWORD, and TOCASEI; the converse is SPAN.

BREAK and SPAN are often used in combination to scan words or other tokens as shown in Figure 30 on page 20. Here is a simpler example:

```
'pattern',
   '(span(" ") or null)',  /* Strip leading              */
   '(break(" ") or rem) ...
```

This scans the first blank-delimited word in the record. Note the use of REST in case BREAK fails because there is only one word on the line. The way shown of skipping leading blanks is slightly faster than using a separate *strip* stage.

This example deletes the beginning of each line up to the first vowel.

Input File:	Pattern:	Output File:
`:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.`	`break("aeiou") rem:out`	`is file is a ample for :hp1.pattern:ehp1.. ee :Figref refid=samp. for ore information.`

CANCEL (Integer)

pattern processing is terminated with a return code and no more input is read. The return code is the integer argument.

CANCEL can be used when it is determined that input is not as expected, or to halt processing at a selected record. The selection stage *tolabel* can be formulated using CANCEL(0), for instance to stop at the first record beginning with “abcd”:

```
pattern "abcd" cancel(0) or rem:out
```

The record stays “in the pipe” where it can be read by a control stage. (This is only relevant when the pipeline containing *pattern* is issued with the CALLPIPE command.)

An EOF pseudo-atom is ignored when CANCEL causes processing to be terminated.

This example copies records up to the first one with a highlight phrase 1 tag.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	eof(output('EOF')) to(":hp1.") cancel(0) or rem:out	:p.This file is a

CASEI (String)

The string is compared with data from the cursor position onwards. String and data bytes are translated to uppercase as per the translate table pointed to by NUCUPPER before being compared.

See also AFTERCI and TOCASEI.

This example selects lines beginning with the word “see”, ignoring case, and copies the rest of the line to the output.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	casei("see ") rem:out	:Figref refid=samp. for

EOF (String)

The string expression is evaluated once after end-of-file is received on input; the result is discarded. This atom must be first in the pattern or after a TOF atom; it cannot be iterated.

To count records:

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	tof(n:=0) eof(output(n ' records')) null(n:=n+1)	4 records

FAIL

Always fails. Iterations and alternatives are tried. This can be used to re-scan a string where things may occur in any order, as shown in this example courtesy of Brent Longborough:

```
To(":nick.") ":nick." (Break(":") Or Rem) :nick Fail Or
To(":node.") ":node." (Break(":") Or Rem) :node Fail Or
Output(Left(nick,8) " " Left(node,8))
```

This example writes the beginning of the line up to “file” as a separate line. All input lines are copied to the output.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	to("file"):out flush fail or rest:out	:p.This :p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.

FENCE

Matches the null string first time it is met. It forces back-up not to try alternatives for the level of expression the fence is at. This is used to say that there is no point in trying alternatives because they are all known not to match.

```
pattern
    "user " fence "abc"
  or
    "dept " ...
```

Here, there is no point in matching what is known to be “user” with some other string when it turns out that the user is not the one wanted.

FENCE is also used in an if/then/else construct to force a failure in the “then” clause to cause a failure of the expression as a whole. Without the fence, a failure in the first part would cause the alternative to be evaluated.

In the following somewhat contrived example, the third record does not have the string “file” in it and thus TO fails; without the fence, the third record would be listed as even.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	null(recno:=recno+1) (if(recno//2) fence to("file"):out or output("Even record " recno))	:p.This Even record 2 Even record 4

FIRST

Matches the null string while the first record of the file is being processed. Fails for all subsequent input records.

FLUSH [(String)]

Matches the null string. The output buffer is flushed as a side effect. The argument (if present) specifies the stream to receive the record; a null expression (assumed when no parentheses are coded) selects the primary stream.

A numeric result is taken as the stream number; a non-numeric result is a stream identifier.

The output stream specified remains selected until the next FLUSH atom is matched or to the next call to the built-in function.

```
pattern arb:stream rem:out flush(stream)
pattern (len(80):out flush or null) rem:out
```

The first example shows how to send each record to the stream specified in the first position; the stream number is not copied to the output. The last example shows how to split a record once after 80 bytes if it is longer than 80 bytes.

The example below performs the same function as *fblock* 10. It is important that the assignment to the output is inside an expression which is being iterated. The second example shows how to write a record for each iteration.

Input File:	**Pattern:**	**Output File:**
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	((len(10) or rem):output)* flush rpos(0)	:p.This fi le is a sample for :hp1.patt ern:ehp1.. See :Figre f refid=sa mp. for more infor mation.
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	(len(10) or rem)*:output flush rpos(0)	:p.This fi :p.This file is a sample for sample for :hp1.patt sample for :hp1.pattern:ehp1.. See :Figre See :Figref refid=sa See :Figref refid=samp. for more infor more information.

IF (Integer)

Matches the null string if the argument is non-zero. Causes a failure if the argument is zero.

```
if(abbrev('abcd','ab'))    /* matches                    */
if(0)                      /* fails                      */
if(-1)                     /* matches                    */
```

If/then/else constructs can be made with parentheses and the OR keyword:

```
(if(<expression>) fence <then> OR <else>)
```

Code NULL when you have no else part.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	if(pos("file", input())) rem:out	:p.This file is a

LEN (Integer)

Matches the number of characters specified by the argument. There must be at least that many characters after the cursor.

LEN can be used to skip over a string found with TO and TOCASEI. A more exotic use is this example where as many characters are skipped as are in a (possibly null) run of stars. The two lines are equivalent; the second one is preferable when there are *many* stars in the run since they do not have to be copied to a variable.

```
(span("*") or null):w len(length(w))
```

```
null(beg:=cursor()) (span("*") or null) len(cursor()-beg)
```

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	len(4):out	:p.T samp See more

NEXTWord [(Enumerated)]

Never fails. Match the null string in the right margin. Match characters up to the first occurrence of a character that is not in the argument string. NEXTWORD matches blanks when the argument is omitted.

NEXTWORD is equivalent to

```
(toword or null)
```

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	nextword("potatochips"):out output("*")	* sa* * *

NOT (String)

The string is compared with data from the cursor position onwards. The null string is matched if the argument is longer than the remaining part of the record or if the string is not found at the cursor position.

The converse is STRING; see also NOTCASEI.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	not(':p.') rem:out	sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.

NOTANY (Enumerated)

A single character is matched if it does not occur in the argument string.

The converse is ANY. See also NOT.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	notany(":p.s") rem:out	ee :Figref refid=samp. for ore information.

NOTCASEI (String)

The null string is matched if there remain fewer input characters than are present in the argument, or if one of the characters is not equal to the corresponding one in the string, ignoring case. String and data bytes are translated to uppercase as per the translate table pointed to by NUCUPPER before being compared.

The converse is CASEI. See also NOT.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	notcasei("s") rem:out	:p.This file is a more information.

NULL [(String)]

Matches the null string. A string expression may be coded in parentheses to set variable(s) as a side effect; the result is discarded.

The first two lines in the example below are equivalent; the third line matches an expression if it is present.

```
(null or (<expression>))
(<expression>)?
((<expression>) or null)
```

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	(to("file"):out or null) rem:r output(translate(r))	:p.This FILE IS A SAMPLE FOR :HP1.PATTERN:EHP1.. SEE :FIGREF REFID=SAMP. FOR MORE INFORMATION.

ONCE (String)

Matches the null string. The string expression is evaluated and the result discarded first time this atom is matched. The argument is ignored forever after.

This atom can be used to evaluate a constant expression or an expression containing the date or time which should be constant for a particular run of *pattern*. A pattern can have any number of ONCE atoms; they need not be matched on the first record.

```
pattern once(time:=time()) '1' out('Printed ' time) or ...
```

Code references to the time function if you wish to timestamp each record as it is processed.

See also TOF and EOF.

In the following example, OUTPUT is a built-in function, not the atom.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	once(output("Once no " recno:=recno+1))	Once no 1

OUTput (String)

Matches the null string. The argument string is appended to the contents of the output buffer as a side effect.

See also FLUSH.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	output("Record " n:=n+1)	Record 1 Record 2 Record 3 Record 4

POS (Integer)

Matches the null string if the cursor is at the position indicated by the number. POS(0) is in the left margin of the input record.

See also RPOS.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	to("f"):out pos(7) rem:out	sample for :hp1.pattern:ehp1.. more information.

REM

Matches all remaining data. This is an infinitesimally faster convenience for RTAB(0). REST is a synonym for REM.

REM is useful in combination with BREAK where a trailing delimiter is to be assumed.

```
... (break(" ")',          /* Skip to end of word         */
    'or rem)...            /* Or take rest                */
```

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	rem:out	:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.

REST

Synonym for REM which matches to the end of the input record.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	rest:out	:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.

RPOS (Integer)

Matches the null string if the cursor is at the position indicated by the number, relative to the end of the input record. RPOS(0) is in the right margin of the input record.

RPOS(0) is often used to force iteration of an expression until the right margin is met. It is important to ensure (often with REM) that the right margin is reached eventually. To delete all occurrences of “abc”:[1]

```
'pattern',
   '(',
      'to("abc"):out',     /* Get to the string          */
      'len(3)',            /* skip it                    */
   'or rem:out',           /* copy rest                  */
   ')* rpos(0)|'           /* force iteration            */
```

See also POS.

To select lines having a single period which is at the end:

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	after("."):out rpos(0)	more information.

RTAB (Integer)

Matches from the cursor position to the position specified relative to the right margin. RTAB fails if the cursor is to the right of the position indicated. RTAB(0), REM, and REST are equivalent.

To align block comment in a REXX program without continuation:

```
'pattern',
   '"/*",                  /* Comment at all?            */
   'rtab(2):line',         /* Take text                  */
   '"*/"',                 /* ending ?                   */
      'out("/*" left(line,68) "*/"', /* write comment     */
   'or rem:out|'           /* Not comment                */
```

See also TAB.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	rtab(3):out	:p.This file i sample for :hp1.pattern:ehp See :Figref refid=samp. more informati

[1] This demonstrates that the replacement operator in SNOBOL4 is not implemented.

SPAN [(Enumerated)]

Characters are matched from the cursor to the right margin or the first occurrence of a character not in the argument string, whichever occurs first. Blanks are spanned when no argument is coded. SPAN fails if the character at the cursor position is not in the string of enumerated characters. The converse is BREAK.

To ensure a record contains only characters from a given alphabet:

```
'pattern',
   '(span(<alphabet>) rpos(0)):out|'
```

Note: SPAN and RPOS are considered an expression; only when both of them match is the record copied to the output. SPAN fails when the first character is not in the alphabet; RPOS fails if SPAN does not reach the right margin.

See also NEXTWORD.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	span("pompously sage"):out	sample mo

STRing (String)

The string is compared with data from the cursor position onwards. The cursor is positioned after the string when a match occurs. The keyword is optional when the expression consists of a single literal or variable reference.

The converse is NOT. See also CASEI.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	string("sample"):out output("<<<" n:=n+1)	sample<<<1

TAB (Integer)

The string from the cursor to the position indicated is matched. TAB fails if the cursor is to the right of the position coded or if the position is beyond the end of the record.

See also RTAB.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	tab(5):out	:p.Th sampl See : more

TO (String)

Matches characters up to (but not including) the argument string. TO fails if the argument string does not occur to the right of the current cursor position.

See also BREAK, TOCASEI, and WORD.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	to("fi"):out	:p.This See :Figref re

TOCASEI (String)

Works like TO, but ignores the case of the argument and the record. String and data bytes are translated to uppercase as per the translate table pointed to by NUCUPPER before being compared.

See also TO, BREAK, and CASEI.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	tocasei("fi"):out	:p.This See :

TOF (String)

The string expression is evaluated once before the first record is read on input; the result is discarded. This atom must be first in the pattern; it cannot be iterated.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	tof(output("At tof:")) rem:out	At tof: :p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.

TOWORD [(Enumerated)]

Characters are matched up to the first occurrence of a character not in the argument string. Blanks are spanned when no argument is coded. TOWORD fails in the right margin.

This atom is similar to SPAN; the difference is that TOWORD matches the null string if the character at the cursor position is not in the argument string; SPAN fails in this case. These two expressions are equivalent:

```
toword(<string>)
(span(<string>) or if(cursor()¬=length(input())))
```

See also NEXTWORD.

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	toword(": ") rem:out	p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.

WORD [(Enumerated)]

Stream to the end of a word delimited by characters in the argument string, or the end of the input if no such character is met. A blank-delimited word is isolated when WORD is coded without an argument.

WORD fails in the right margin; it also fails if the character at the cursor position is in the argument string.

This atom is similar to BREAK, with the difference about the handling of the last word on a line. These two expressions are identical:

```
word(<string>)
if(cursor()¬=length(input()))
  (notany(<string>) (break(<string>) or rest))
```

Input File:	Pattern:	Output File:
:p.This file is a sample for :hp1.pattern:ehp1.. See :Figref refid=samp. for more information.	word:out	:p.This sample See more

String Expressions

A string expression has the syntax and semantics of a REXX expression at the language level corresponding to VM/System Product Release 6, with these differences and restrictions:

- Arithmetic is integers only, done with 32 bit precision in 2-complement form. Hence, there is no divide operator (/); integer divide and integer remainder are provided. Conversion between string and integer is performed as required.
- Composite operators must be written without white space between the characters. That is, "= =" is incorrect.
- The null string converts to zero.
- White space means abuttal without a blank. That is, the blank operator acts like || in REXX; no operator catenates with blank.
- The precedence of the abuttal operator is different than the REXX blank operator.
- The backslant (code point `X'E0'`) is a national character that can be used in a name; it cannot be used as a prefix to negate the effect of a relational operator.

Some built-in functions are equivalent to REXX; others are unique to *pattern*.

White Space

White space consists of

- Blanks
- Comments
- `%include` statements
- End-of-line in included files.

Additional Operators

Two operators are defined that do not exist in REXX.

Assignment

The colon-equal operator (:=) assigns the right-hand side to the variable defined by the left-hand side.

Note that colon-equal is an operator; it may occur more than once in an expression and it can be in the middle of the expression. The result of the assignment operator is the value assigned to the variable.

```
null(c:=b:=a)
```

The precedence of the assignment operator is between plus/minus and abuttal on the right-hand side; on the left-hand side, the colon-equal operator has the highest precedence of the binary operators. The prefix operators have higher precedence than the assignment operator.

Figure 36. Integer Expression Sample

```
|pattern output('Record ' line:=line+1 ':') rem:out|
```

Here the record number is counted in the variable “line”. It starts out being the null string which is the same as zero, and is incremented for each record.

Separator

The semicolon operator causes evaluation of the expressions on each side, the left-hand one first. The result is the left-hand side; the right-hand expression is evaluated only for its side effects. For example, to assign a and b the values 1 and 2, respectively:

```
null(a:=1; b:=2)
null(a:=1; b:=a+1)
```

The semicolon looks like a statement separator, but it separates expressions, not statements (or instructions). The semicolon operator has the lowest precedence of all operators.

Operator Precedence

Operator precedence different from REXX is shown in Figure 37 where each half of a line represents identical expressions.

Figure 37. Operator Precedence

```
c var2:=var1:=a+b c(var2:=(var1:=(a+b)))
var2:=1+var1:=a+b var2:=(1+(var1:=(a+b)))
a+b c*d  (a+b)(c*d)
```

Built-in Functions as in REXX

The following functions have, within the 32 bit integer precision, syntax and semantics as defined for the REXX functions with the same name. Only eight characters are used to determine the name, and it may be in any case; “translating” is equivalent to “TRANSLAT.”

```
ABBREV ABS BITAND BITOR BITXOR B2C CENTER CENTRE COMPARE COPIES C2B C2D C2X
DATE DATATYPE DELSTR DELWORD D2C D2X FIND INDEX INSERT JUSTIFY LASTPOS LEFT
LENGTH MAX MIN OVERLAY POS REVERSE RIGHT SIGN SPACE STORAGE STRIP SUBSTR
SUBWORD TRANSLATe VALUE VERIFY WORD WORDINDEx WORDLENGth WORDPOS WORDS
XRANGE X2C X2D
```

These functions have a restricted range of arguments; they do give the same result as does REXX for defined arguments.

TIME Supported options are none, Normal, Civil, Hours, Minutes, and Seconds. No further resolution is available from Diagnose C, which is used to obtain the local time of day.

The eight-byte TOD clock value is returned when the argument is TOD.

DATE and TIME are based on the result from Diagnose C where CP has performed the time-zone adjustment. The diagnose is invoked once only for each

evaluation of an expression (atom) where either of the functions is used; performance may be improved if the value is saved in a variable using ONCE.

Built-in Functions Particular to *pattern*

These functions have no counterpart in REXX.

CMS - Issue CMS Commands

One string argument is required. The argument is sent to the CMS subcommand environment for execution. The result is the integer return code.

CURSOR - Return the Cursor Position

One argument is optional. The cursor position before matching the atom where the function is referenced.

This position is relative to the left margin when no argument is coded. When an argument is coded, the first character is converted to uppercase and inspected:

A "Absolute". The absolute storage address (expressed as an integer) is returned.

R "Remainder". The cursor position relative to the right margin is returned.

C2F - Format an 8-byte Character String as a Floating Point Number.

The result is a string of length up to 22. The last digit is not exact with very large exponents.

FLUSH - Flush the Output Buffer into the Pipeline

One argument is optional. Flushes the output buffer in the same way done by the atom of the same name.

The result is a null string if the record is written successfully; it is "1" if the output stream is not connected. It is an error if the output stream is not defined. FLUSH is useful in EOF. The output stream specified remains selected until the next FLUSH atom is matched or to the next call to the built-in function.

F2C - Convert a Floating-point Number to the Internal Representation for a Long Floating-point Number.

The result is an 8 byte character string. It is an error if the number cannot be converted.

INPUT - Return Contents of the Current Input Record

Up to three arguments may be used; none are required. The arguments are similar to the arguments to SUBSTR.

1. The first position to return. This must be a positive number; the default is the beginning of the input record.
2. The number of characters to return. This must be a number which is zero or more. The default is to return up to the end of the input record.
3. A pad character to use when the length requested takes the result beyond the end of the input record.

OUTBUF - Return Contents of the Output Buffer

Up to three arguments may be used; none are required. The arguments are similar to the arguments to SUBSTR.

1. The first position to return. This must be a positive number; the default is the beginning of the output buffer.
2. The number of characters to return. This must be a number which is zero or more. The default is to return up to and including the last character loaded in the output buffer so far.
3. A pad character to use when the length requested takes the result beyond the last character loaded in the output buffer.

OUTPUT - Append to the Output Buffer

Three arguments are optional. If present the second argument must evaluate to a non-negative number; the output buffer is padded to the length specified if the contents is not already longer than the length specified. The third argument specifies the pad character to use; the default is a blank. The first argument string is then appended to the contents output buffer as done by the atom of the same name. The result is a null string. OUTPUT is useful in EOF.

PIPCMD - Issue a Pipeline Command

One argument is required. The subcommands available are listed in in *Pipeline Subcommands* in the User's Guide (*Pipeline Commands* in PIPUG LIST3820). The most useful one is no doubt CALLPIPE. The integer result is the return code from the command. Note that -7 means that the command is not recognised.

SETDATE - Set the Date and Time to Use

One 16-byte argument is required. The date and time used in the remainder of the expression are set to the value of the argument string. The argument string is in the format stored by diagnose 0: mm/dd/yyhh:mm:ss. This function is provided to test the DATE and TIME functions in repeatable circumstances which tends to give more confidence that for instance leap years are handled correctly.

SUBCOM - Issue a String to a Subcommand Environment

Two arguments are required. The second argument is addressed to the subcommand environment named by the first argument. The result is the integer return code from the subcommand environment. The subcommand name is translated to uppercase and padded with blanks or truncated to eight characters.

REXX Functions not Provided

The following built-in functions are not present:

ADDRESS	No default command environment is established. Use CMS or SUBCOM functions.
ARG	Functions are not defined; hence there are no arguments.
CMSFLAG	System-dependent.
DATATYPE	
DIAG/DIAGRC	System-dependent.
ERRORTEXT	Implementation dependent.
EXTERNALS	System-dependent.

FORMAT	Fractional numbers are not implemented.
LINESIZE	Implementation-dependent
QUEUED	System-dependent.
RANDOM	
SOURCELINE	Implementation dependent.
SYMBOL	Implementation dependent.
TRACE	Implementation dependent.
TRUNC	Fractional numbers are not implemented.
USERID	System-dependent.

Messages

199E Excessive symbol table entry size <bytes>

Explanation: A module requested a symbol table too large.

System Action: The stage terminates with return code 199.

User Response: Contact your system support staff.

System Programmer Response: This is an error in the program calling the symbol table module.

201E Range missing

Explanation: An equal sign (=) is coded in a pattern indicating an explicit range of repetitions, but the decimal data is not found.

System Action: The stage terminates with return code 201.

202E Variable name missing

Explanation: A character is coded indicating that a variable should be set or referenced, but no more data follows.

System Action: The stage terminates with return code 202.

203E Pattern missing

Explanation: An expression is opened with a left parenthesis or an OR, but nothing is met before the closing right parenthesis.

System Action: The stage terminates with return code 203.

205E Atom parameter missing

Explanation: An atom that requires a parameter is not followed immediately by a left parenthesis.

System Action: The stage terminates with return code 205.

207E Alternative missing

Explanation: An expression is opened with a left parenthesis or an OR, but nothing is met before the following OR.

System Action: The stage terminates with return code 207.

208E Cursor does not advance from position <number> in record <number>

Explanation: A pattern item is matched repeatedly without advancing the cursor.

System Action: The stage terminates with return code 208.

User Response: Inspect the contents of the record to see what causes a null string to be matched repeatedly. The error is easily provoked:

```
208; literal a|pattern null* rpos(0)
►PIPSNO208E Cursor does not advance from position
►PIPDSP020I Pipeline  stage 2 returned with code 2
```

213E Null string for atom

Explanation: An atom requiring an enumerated list of characters has a null argument (length zero).

System Action: The stage terminates with return code 213.

221E Invalid data "<token>" in expression

Explanation: The first character is not valid; it must be a number, a letter, a national character, or one of the quotes.

System Action: The stage terminates with return code 221.

User Response: A cause can be that a string is specified with a delimiter character that is not a single or double quote. A string must be enclosed in single or double quotes; you cannot use the standard &cms. of any delimiter character.

System Action: The stage terminates with return code 221.

239I Old-style pattern syntax

Explanation: The pattern is likely to become invalid in the next edition of *CMS Pipelines*.

System Action: Message 192 is issued if the message level is odd. Processing continues.

User Response: Change the pattern to a modern syntax.

240E Function <name> not supported

Explanation: A pattern expression has an identifier followed by a left parenthesis, indicating a function call, but the function requested does not exist.

System Action: The &stg. terminates with &rtrncd. .

242E Too few arguments; <number> is minimum

Explanation: Too few arguments were present for the function call.

System Action: The stage terminates with return code 242.

243E Too many arguments; <number> is maximum

Explanation: Too many arguments were present for the function call.

System Action: The stage terminates with return code 243.

244E Error in call to <function>

Explanation: The function indicates an error in its invocation.

System Action: The stage terminates with return code 244.

User Response: This is often caused by an argument that is not within an acceptable range or an overflow in the computation. This message is issued, for instance, when the second argument to LEFT is negative.

246E Argument cannot be omitted

Explanation: A function argument is left out, but the function in question does not support this.

System Action: The stage terminates with return code 246.

247E Unsupported conversion

Explanation: Conversion is attempted between two formats that are not compatible.

System Action: The stage terminates with return code 247.

248E Conversion error; "<string>" not integer

Explanation: The string shown cannot be converted to an integer.

System Action: The stage terminates with return code 248.

249E Unexpected comma

Explanation: A comma is met in an expression in the argument for an atom.

System Action: The stage terminates with return code 249.

250E Syntax error in expression

Explanation: A pair of parentheses are met with nothing except white space inside.

System Action: The stage terminates with return code 250.

251E Unknown operator "<string>"

Explanation: The string consists of characters that can be used in an operator, but no operator is known as shown.

System Action: The stage terminates with return code 251.

252E Operand missing

Explanation: A binary operator is found, but one of its operands is missing.

System Action: The stage terminates with return code 252.

254E Delimiter missing after string in "<atom type>" beginning "<string>"

Explanation: A string argument to an atom is not properly delimited.

System Action: The stage terminates with return code 254.

259E Invalid hex data "<string>"

Explanation: Valid hexadecimal data consists of one or more characters in the ranges a-f, A-F, and 0-9. An odd number of characters is padded with a zero on the left. Blank characters may be present at byte boundaries.

System Action: The stage terminates with return code 259.

260I Doing pattern item <number> near "<string>"

Explanation: Information message issued when pattern matching stops due to a runtime error.

262E Expression already met for <which> or it is not first

Explanation: A duplicate TOF or EOF expression is met or some other atom is met before this atom.

System Action: The stage terminates with return code 262.

User Response: Note that only one of each of the atoms is allowed, and that they must be first in the pattern specification.

263E Atom "<atom>" not supported

Explanation: A left parenthesis is found immediately after an identifier, but the identifier is not a known atom.

System Action: The stage terminates with return code 263.

User Response: Put a blank character between a variable reference and a parenthesis opening an expression.

265W Atom argument should be in parentheses

System Action: This is a warning message. Processing continues with a single string or variable as the argument to the atom.

User Response: Put the argument to the atom in parentheses to be compatible with the future.

266E Ampersand no longer allowed to identify a variable

Explanation: The usage of an ampersand (&) to designate a variable is no longer supported. The ampersand indicates the logical and operator in a string expression.

System Action: The stage terminates with return code 266.

User Response: Remove the ampersand and enclose the variable reference in parentheses.

267E Number <number> not whole

Explanation: A function which requires an integer argument receives an argument which is not integer.

System Action: The stage terminates with return code 267.

268W There should be no blanks between the atom and the parentheses with the argument

Explanation: Blank characters are found between an atom name and the parentheses containing its argument.

System Action: This is a warning about down-level syntax. Processing continues.

User Response: Abut the parentheses to the atom. Correct the pattern specification at the first convenient opportunity.

271E The atom <atom> cannot have parameters

Explanation: An opening parenthesis is met immediately after an atom that does not take parameters.

System Action: The stage terminates with return code 271.

User Response: Insert a blank before a parentheses used to group a pattern expression.

272E Missing end of comment

Explanation: A comment opened with "/*" did not have a matching "*/".

System Action: The stage terminates with return code 272.

User Response: Comments can be nested. Ensure your comments are properly ended.

273E Incomplete %include statement

Explanation: No ending parenthesis found.

System Action: The stage terminates with return code 273.

User Response: Ensure the ending parenthesis for the %include statement is on the same line as the beginning of the statement.

274E File name missing

Explanation: No file name is found for %include.

System Action: The stage terminates with return code 274.

275E %include recursion

Explanation: A file is to be imbedded which is already in the process of being imbedded.

System Action: The stage terminates with return code 275.

276E %include file not found

Explanation: It is not possible to read from the file identified in a %include order.

System Action: The stage terminates with return code 276.

277E Too many %include files

Explanation: The information concerning files included is too big to handle.

System Action: The stage terminates with return code 277.

278I <...> line <number> of file "<file>"

Explanation: Informational message for syntax errors in included files.

Appendix A. Miscellaneous Reference Information

Formal Syntax of a Pattern

Figure 38 shows the formal syntax of a pattern specification. <s-expr> means a string expression defined in "String Expressions" on page 42; <delimitedString> is defined in *CMS Pipelines User's Guide*.

Definitions marked /*?*/ are more likely to change or be removed than the ones not so marked. The comment string is not part of the syntax definition.

Figure 38. pattern Syntax

```
<pattern>    ::= [TOF(<s-expr>)] [EOF(<s-expr>)] <p-expr>
<p-expr>     ::= <list>|<list> OR <p-expr>
<list>       ::= <item>|<item> <list>
<item>       ::= <a-expr>[<iteration>][:<assignment>]
<a-expr>     ::= <atom> | (<p-expr>)
<iteration>  ::= ? | * | + | =<number>[-<number>]
<assignment> ::= <variable> | (<s-expr>)

<atom> ::= <identifier>(<s-expr>) |
           <variable> |                                    /*?*/
           <qstring> |
           <delimitedString>                               /*?*/

<qstring> ::= <qqstr> | <qstr>X
<qqstr>   ::= <qstr> | <qstr><qqstr>
<qstr>    ::= "<string>" | '<string>'

<variable> ::= <identifier>

<identifier> ::= {_ | <letter>} ffl_ | <letter> | <digit>"...
```

A quoted string may contain two adjacent quotes of the type used to delimit the quoted string. This represents a single quote in the string. Hexadecimal data are indicated by an x after the ending quote. Strings delimited by other characters than single and double quote are not scanned for this.

Notes for SNOBOL4 Gurus

pattern is not an attempt to re-implement SNOBOL4. However, because of the similarities, the differences might be overlooked, so:

- Pattern expressions are more or less equivalent though *pattern* has more syntactic sugar and accepts mixed case names. Argument expressions are different.
- Pattern matching is like FULLSCAN on and ANCHORED mode.
- Variable assignment is immediate; conditional assignment to a variable is done with a subterfuge. To assign v conditionally in <pattern>, code some other variable name (e.g., vi) as an immediate assignment. Then,

```
(<pattern>) null(v:=vi)
```

performs the same function as the conditional assignment.

- Variables assigned a value from the input record through matching (as opposed to being set in a string expression) lose their contents when the next record is read. Use the conditional assignment subterfuge shown above to overcome this.
- There is no replacement statement. Its function can be implemented with OUTPUT.
- The *pattern* atom ARB is equivalent to LEN(1) which is not the same as the SNOBOL4 definition of ARB. SNOBOL4 ARB is coded as ARB* in *pattern*.

Considerations for Compatibility with the Future

- At the moment, a string not inside parentheses can be delimited by any *special* character. However, single and double quotes should be used to delimit strings; they are mandatory in `%include` files and in string expressions. Parentheses and ampersand are not accepted; REXX operators are discouraged.
- Do not depend on truncation of variable names and function names.
- Do not depend on the uppercasing of variable names.

Compatibility with the Past

- The argument to an atom is in parentheses in general, but a few special cases could be written in a shorter notation. We are in the process of removing these; the usage attracts a warning message.
  - A number may be coded without the parentheses when the atom is listed as requiring an integer argument.

Incompatibilities with the Past

- Usage of & to identify a variable parameter to an atom (when not in parentheses) is no longer valid. In a string expression it is now the logical and operator.
- Hex strings are supported. Earlier this was interpreted as the character string abutted a reference to the variable `X`.
- A quote-delimited string can contain two quotes to indicate a single quote is desired. Earlier this was interpreted as two abutted strings.
- Assignment in a string expression can no longer be done with the colon operator.
- The first 250 characters of variable names are significant. Programs that relied on truncation after eight characters will fail.

Appendix B. National Characters

Figure 39 shows the graphics for the national use code points on a 3270.

Figure 39. National Characters

US	UK	DK	Hex	What you see on a US terminal:
#	#	Æ	7B	Number sign.
@	@	Ø	7C	At sign.
$	£	Å	5B	Dollar sign.
{	{	æ	C0	Left brace.
¦	¦	ø	6A	Broken bar.
}	}	å	D0	Right brace.
\	\	\	E0	Back slash (backslant).
¢	$	#	4A	Cent sign.
!	!	¤	5A	Exclamation sign.
~	¯	ü	A1	Tilde.
`	`	`	79	Accent grave.

Appendix C. Sample Traces

This section contains the regression test output from running a few patterns, showing how things are matched. Trace data are written to the same stream as normal output data.

Figure 40 shows a trace of a pattern which does not iterate.

Figure 40. Sample Trace

```
pipe literal a b c d e f|.trc:pattern ("a b c" rem):out|cons
```

```
Begin input record:
a b c d e f
Matching item number        2
Matching STRING   cursor at        0 up to 8 bytes data:a b c d
a b c
Matched.  Cursor at        5
a b c
Going on to brother.
Matching item number        3
Matching REM      cursor at        5 up to 8 bytes data: d e f
Matched.  Cursor at       11
 d e f
Setting OUT
a b c d e f
Outputting:
a b c d e f
a b c d e f
```

The way an expression is iterated is shown in Figure 42 on page 55. Note that the parentheses are skipped first time since the asterisk means match zero or more times.

Figure 41 shows the pipeline run to obtain the trace.

Figure 41. Program to Generate Sample Trace

```
/* Create sample trace */
'PIPE literal a   record  |',
   'x.trc:pattern',
      '(break(" "):out span(" ") out("*") or rem:out)* rpos(0)|',
   '> sample trace a'
```

Figure 42 (Page 1 of 2). More Sample Trace

```
Begin input record:
a   record
Matching item number        6
Matching RPOS     cursor at        0 up to 8 bytes data:a   reco
0
Failure.
Backing up to previous.
Matching item number        2
Matching BREAK    cursor at        0 up to 8 bytes data:a   reco

Matched.  Cursor at        1
a
Setting OUT
a
Going on to brother.
Matching item number        3
Matching SPAN     cursor at        1 up to 8 bytes data:   recor

Matched.  Cursor at        4

Going on to brother.
Matching item number        4
Matching OUTPUT   cursor at        4 up to 8 bytes data:record
"*"
Matched.  Cursor at        4
Going on to brother.
Matching item number        6
Matching RPOS     cursor at        4 up to 8 bytes data:record
0
Failure.
Backing up to previous.
Matching item number        2
Matching BREAK    cursor at        4 up to 8 bytes data:record

Matched.  Cursor at       10
record
Setting OUT
record
Going on to brother.
Matching item number        3
Matching SPAN     cursor at       10 up to 8 bytes data:

Matched.  Cursor at       12

Going on to brother.
Matching item number        4
Matching OUTPUT   cursor at       12 up to 8 bytes data:
"*"
Matched.  Cursor at       12
```

Figure 42 (Page 2 of 2). More Sample Trace

```
Going on to brother.
Matching item number        6
Matching RPOS     cursor at       12 up to 8 bytes data:
0
Matched.  Cursor at       12
Outputting:
a*record*
a*record*
```

Index

Special Characters

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W