IBM InfoSphere QualityStage
Version 8 Release 7

Pattern Action Reference

SC19-3480-00

IBM InfoSphere QualityStage
Version 8 Release 7

Pattern Action Reference

SC19-3480-00

Note

Before using this information and the product that it supports, read the information in "Notices and trademarks" on page 59.

Contents

IBM InfoSphere QualityStage Pattern Action reference

To obtain correct standardization, you need to understand the concepts of pattern matching and the reasons for matching.

This reference material describes the Pattern Action language and pattern-action files. It is for application developers. A pattern-action file consists of a series of patterns and associated actions. Incoming data either matches or does not match a pattern. If the incoming data matches a pattern, then actions associated with the pattern are executed. If the incoming data does not match the pattern, the actions are skipped.

Introduction to the Pattern Action language

You use the Pattern Action language to manipulate data. You can decipher and identify patterns in data, and then perform actions against the data based on the pattern.

A Pattern-Action file contains a series of pattern action sets. Each set contains a pattern condition followed by action statements. Actions are taken against input data that has been separated into tokens and classified. Actions are based on a given pattern of tokens.

The pattern condition can contain the following elements:

Operands
Class representation of the incoming data. The class representation is either a user-defined class or a default class.

User variables
Defined by the user, symbolic names associated with values that can be changed.

Dictionary fields
A collection of output field names as defined in the dictionary table located in the dictionary definition file (.DCT).

Patterns are executed in the order they appear in the Pattern-Action file. A pattern either matches the input data or it does not match. If it matches, the actions associated with the pattern are executed. If not, the actions are skipped. Processing then continues with the next pattern in the file.

The pattern action file enables you to use logic which becomes a part of rule sets. Rule sets, applied against parsed and classified input data, standardize the data.

Pattern-Actions files are ASCII files that you can create or update by using any standard text editor.

Describing the pattern format

A pattern can consist of one or more operands. The operands are separated by vertical lines. For example, the following pattern has four operands:

```
^ | D | ? | T
```

These are referred to in the actions as [1], [2], [3], and [4].

You can use spaces to separate operands. For example, the following two patterns are equivalent:

```
^|D|?|T
^ | D | ? | T
```

Tip: The use of spaces to separate operands and pipes makes reading and debugging the patterns easier. Spaces are used in the prebuilt rules provided with IBM® InfoSphere® QualityStage™.

You can add comments by preceding them with a semicolon. An entire line or just the end of a line can be a comment. For example:

```
;
;Process standard addresses
;
^ | D | ? | T  ; 123 N MAPLE AVE
```

You can refer to fields by enclosing the column name in braces. For example, {HouseNumber}, {StreetPrefixDirectional}, {StreetName}, and {StreetSuffixType} refer to dictionary column names that are defined in the dictionary definition file.

```
^ | D | ? | T | $ |
[ {HouseNumber} = "" & {StreetPrefixDirectional} = "" &
 {StreetName} = "" & {StreetSuffixType} = "" ] ;
 Common Pattern Found: CALL Address_Type SUBROUTINE then EXIT
```

Pattern matching for an individual pattern line stops after the first match is found. For example, in the address 123 MAPLE AVE & 456 HILL PLACE, the following pattern matches to 123 MAPLE AVE but not to 456 HILL PLACE:

```
^ | ? | T
```

A pattern might not contain any operands. For example, the conditional statement `[Required_Step ="TRUE"]` is a pattern that contains a user-variable: *Required_Step*.

The simplest patterns consist of only classification types:

```
^ | D | ? | T
```

These are straightforward and do not require much further explanation. Hyphens and slashes can occur in the patterns. For example:

```
123-45 matches to ^ | - | ^

123 ½ matches to ^ | ^ | / | ^
```

Parsing elements (PRAGMA)

The standardization process begins by identifying tokens within the incoming data. A token can be a single character, a word, or multiple words that are not separated by spaces.

The parsing parameters of the table in the pattern-action file define the tokens. For example, for Latin-based languages, 123-456 has three tokens: 123, the hyphen (-), and 456. A hyphen separates words and is considered to be a token in itself.

Spaces are separate tokens. They are also stripped from the input. For example, 123 MAIN ST consists of three tokens: 123, MAIN, and ST.

Using SEPLIST and STRIPLIST

SEPLIST and **STRIPLIST** are specification statements that are placed between the **PRAGMA_START** and **PRAGMA_END** lines in a Pattern-Action file.

You can override the default assumptions by specifying one or both of the following statements:

- **SEPLIST**. Uses any character in the list to separate tokens
- **STRIPLIST**. Removes any character in the list.

Any character that is in both lists separates tokens but does not appear as a token itself. The best example is spaces. One or more spaces are stripped but the space indicates where one word ends and another begins. Include the space character in both **SEPLIST** and **STRIPLIST**.

If you want to include **SEPLIST** and **STRIPLIST**, put them as the first set of statements in the .pat file, preceded with a **PRAGMA_START**, and followed by a **PRAGMA_END**. For example:

```
\PRAGMA_START
SEPLIST " ,"
STRIPLIST " -"
\PRAGMA_END
```

Enclose the characters in the list in quotation marks.

Applying parsing rules to a list

The special token class (~) represents special characters that are not included in SEPLIST and STRIPLIST. These characters (!, \, @, ~, %) require special handling.

When adding special characters, consider the following rules:

- Do not use the quotation mark in either **SEPLIST** or **STRIPLIST** list unless you precede it with the escape character backslash (\).
- The backslash (\) is the escape character that you use in a pattern but it must itself be escaped (\\).

In this example, the space is in both lists and the hyphen is in the **STRIPLIST** but not the **SEPLIST**. Hyphens are stripped so that STRATFORD-ON-AVON is considered to be STRATFORDONAVON.

```
SEPLIST: " !?%$,.;:()/#&"
STRIPLIST: " !?*@$,.;:-\\''"
```

In this example, the hyphen is in both lists. Because the **SEPLIST** is applied before the **STRIPLIST**, STRATFORD-ON-AVON in the incoming data is parsed into three tokens: STRATFORD, ON, and AVON.

```
SEPLIST: " !?%$,.;:-()/#&"
STRIPLIST: " !?*@$,.;:-\\''"
```

In this example, the comma separates tokens so that the city name and state can be found (SALT LAKE CITY, UTAH). Any other special characters are classified as a special type.

```
SEPLIST: " !?%$,.;:()-/#&"
STRIPLIST: " !?*@$.;:\\''"
```

Each rule set has its own lists. If no list is coded for a rule set, the following default lists are used:

```
SEPLIST: " !?%$,.;:()-/#&"
STRIPLIST: " !?*@$,.;:\\''"
```

When overriding the default **SEPLIST** and **STRIPLIST**, do not cause collisions with the predefined class meanings because the class of a special character changes if it is included in the **SEPLIST**.

If a special character is included in the **SEPLIST** and not in the **STRIPLIST**, the token class for that character becomes the character itself.

For example, ^ is the numeric class specifier. If you add this character to **SEPLIST** and not to **STRIPLIST**, any token consisting of ^ is given the class of ^ . This token would next match to a numeric class (^) in a Pattern-Action file.

Specifying the tokenizer

In the PRAGMA section of the pattern-action file, you can use the **TOK** command to specify the regional setting that you want to use in a rule set, and thereby indicate the way you want tokens handled.

TOK is an optional specification statement. If you do not specify a tokenizer, the tokenizer used is based on the regional setting of the computer on which you run investigation or standardization. You can choose one of the following tokenizers:

Tokenizer	Description
Latin-based	For languages such as English, Spanish, French, and German. Tokens are typically separated by spaces.
CJK	For Chinese, Japanese and Korean languages. Tokens are typically not separated.

The syntax for **TOK** is as follows:

```
TOK locale
```

Locale is the International Components for Unicode (ICU) locale standard.

During standardization, the CJK tokenizer is used if the TOK command is followed by a locale variable that begins with one of the following codes:

- ja
- zh
- ko
- vi

If the locale variable is any other value, the Latin-based tokenizer is used.

For example, if you specify `TOK en_US`, the tokenizer includes Latin-based language considerations in the tokenization approach. If you specify `TOK jp_JP`, the tokenizer includes locale-specific (CKJ) considerations in the tokenization approach.

Unconditional patterns

Unconditional patterns are not sensitive to individual values, they are the simplest to code and the most general. You can specify conditions to cause patterns to match only under specified circumstances.

Identifying simple pattern classes

Simple pattern classes are used to further identify the data with a meaningful pattern from which to match the pattern actions.

Simple pattern classes are represented by single characters.

Within patterns, you must use the escape character backslash (\) to prevent the syntax of the pattern tables from interfering with certain single character classes. Use the escape character backslash (\) with the following single character classes: the hyphen -, slash /, number sign #, left and right parentheses () and ampersand &.

Take care when specifying **SEPLIST** and **STRIPLIST** entries. For example, to recognize the ampersand as a single token, include it in **SEPLIST** but not in **STRIPLIST**. If the backslash is in **SEPLIST**, its class is \ (backslash). If a backslash is used in a pattern, then it must have an escape character in a pattern as a double backslash (\\). Also see "Applying parsing rules to a list" on page 3

The **NULL** class (0) is not included in this list of single character classes. The **NULL** class is used in the classifications (.CLS) or in the **RETYPE** action to make a token **NULL**. Because a **NULL** class never matches to anything, it is never used in a pattern.

The simple pattern classes are as follows:

Table 1. List and description of simple pattern classes

Class	Description
A - Z	User-supplied class from the classifications The classes A to Z correspond to classes that you code in the classifications. For example, if APARTMENT is given the class of U in the classifications, then APARTMENT matches a simple pattern of U.
^	Numeric The class ^ (caret) represents a single number, for example, the number 123. However, the number 1,230 uses three tokens: the number 1, a comma, and the number 230.
?	One or more consecutive words that are not in classifications. The class ? (question mark) represents one or more consecutive alphabetic words. For example, MAIN, CHERRY HILL, and SATSUMA PLUM TREE HILL each match to a single ? class provided none of these words are in the classifications for the rule set. Class ? is useful for street names when multi-word and single-word street names must be treated identically.

+	A single alphabetic word that is not in classifications
	
	The class + (plus sign) is useful for separating the parts of an unknown string. For example, in a name like OWAIN LIAM JONES, copy the individual words to columns with given name, middle name, and family name as follows:
	`+ | + | +`
	`COPY [1] {GivenName}`
	`COPY [2] {MiddleName}`
	`COPY [3] {FamilyName}`

Table 1. List and description of simple pattern classes (continued)

Class	Description
&	A single token of any type The class & (ampersand) represents a single token of any class. For example, a pattern to match to a single word following an apartment type is: `U | &` `SUITE 11` is recognized by this pattern. However, in a case such as `APT 1ST FlOOR`, only `APT 1ST` is recognized by this pattern.
\&	Type the escape character backslash (\) in front of the ampersand to use the ampersand as a literal. `< | \& | ? | T` 1ST & MAIN ST is recognized by this pattern.
>	Leading numeric The class > (greater than symbol) represents a token with numbers that is followed by letters. For example, a house number like 123A MAPLE AVE can be matched as follows: `> | ? | T` `123A` is recognized by this pattern. The token contains numbers and alphabetic characters but the numbers are leading. In this example, `T` represents street type.
<	Leading alphabetic character The class < (less than symbol) matches itself to leading alphabetic letters. It is useful with the following examples: A123 ALPHA77 The token contains alphabetic characters and numbers but the alphabetic characters are leading.
@	Complex mix The class @ (at sign) represent tokens that have a complex mixture of alphabetic characters and numerics, for example: A123B, 345BCD789. For example, area information like Hamilton ON L8N 2P1 can be matched as follows: `+ | P | @ | @` In this example, P represents Province. The first @ represents `L8N` and the second @ represents `2P1`.
~	Special punctuation The class ~ (tilde) represents special characters that are not in the **SEPLIST**. For example, if a **SEPLIST** does note contain the dollar sign and percent sign, then you might use the following pattern: `~ | +` In this example, $ HELLO and % OFF match the pattern.

Table 1. List and description of simple pattern classes (continued)

Class	Description
/	Literal The class / (slash) is useful for fractional addresses like 123 ½ MAPLE AVE, which matches to the following pattern: `> | ^ | / | ^ | ? | T`
\/	Backslash, forward slash You can use with the escape character backslash (\) with the slash in the same manner you use the / (slash) class.
-	Literal The class - (hyphen) is often used for address ranges, for example, an address range like 123-127 matches the following pattern: `^ | - | ^`
\-	You can use with the escape character backslash (\) with the hyphen in the same manner you use the - (hyphen) class.
\#	Literal. You must use with the escape character backslash (\), for example: \#. The class # (pound sign) is often used as a unit prefix, for example, an address like suite #12 or unit #9A matches the following pattern: `U | \# | &`

()	Literal
	
	The classes (and) (parentheses) are used to enclose operands or user variables in a pattern syntax. An example of a pattern syntax that includes a leading numeric operators and a trailing character operator is as follows:
	`> | ? | T`
	`COPY [1](n) {HouseNumber}`
	`COPY [1](-c) {HouseNumberSuffix}`
	`COPY [2] {StreetName}`
	`COPY_A [3] {StreetSuffixType}`
	`EXIT`
	
	The pattern syntax example, can recognize the address 123A MAPLE AVE. The numbers 123 are recognized as the house number and the letter A is recognized as a house number suffix.
	
	Use with the escape character backslash (\) with the open parentheses or close parentheses to filter out parenthetical remarks. To remove a parenthetical remark such as (see Joe, Room 202), you specify this pattern:
	`\( | ** | \)`
	`RETYPE [1] 0`
	`RETYPE [2] 0`
	`RETYPE [3] 0`
	
	The code example removes the parentheses and the contents of the parenthetical remark. In addition, when you retype these fields to **NULL** you essentially remove the parenthetical statement from consideration by any patterns that are further down in the Pattern-Action file.
	
	The **NULL** class (0) is not included in this list of single character classes. The **NULL** class is used in the classifications or in the **RETYPE** action to make a token **NULL**. Because a **NULL** class never matches to anything, it is never used in a pattern.

Table 1. List and description of simple pattern classes (continued)

Class	Description

\(and \)	Use with the escape character backslash (\) with the open parentheses or close parentheses to filter out parenthetical remarks. To remove a parenthetical remark such as (see Joe, Room 202), you specify this pattern:
	`\( | ** | \)`
	`RETYPE [1] 0`
	`RETYPE [2] 0`
	`RETYPE [3] 0`
	
	The code example removes the parentheses and the contents of the parenthetical remark. In addition, when you retype these fields to **NULL** you essentially remove the parenthetical statement from consideration by any patterns that are further down in the Pattern-Action file.

Applying subfield classes (1 to 9, -1 to -9)

The subfield classes 1 to 9, and -1 to -9 are used to parse individual words of an ? string.

The number 1 represents the first word, the number 2 represents the second, the number –1 represents the last word, and the number –2 represents the next to last word. If the word referenced does not exist, the pattern does not match. If you are processing company names and only wanted the first word, a company name like WILLIAMS BIG SUPER CELL COMPANY matches to the following patterns (assume COMPANY is in the classifications (.CLS) as a type C).

+ | + | + | + | C	WILLIAMS is operand [1], BIG is operand [2], SUPER is operand [3] CELL is operand [4] and COMPANY is operand [5]
? | C	WILLIAMS BIG SUPER CELL is operand [1], COMPANY is operand [2]
1 | C	WILLIAMS is operand [1], COMPANY is operand [2]
2 | C	BIG is operand [1], COMPANY is operand [2]
–1 | C	CELL is operand [1], COMPANY is operand [2]
–2 | C	SUPER is operand [1], COMPANY is operand [2]

You can combine single alphabetic classes (**+**) with subfield classes. For example, in a series of consecutive unknown tokens like CHERRY HILL SANDS, the following pattern causes the following match:

```
+ | -1
```

The **+** matches to the word CHERRY and the **–1** matches to SANDS. The operand [1] is CHERRY and operand [2] is SANDS.

Specifying subfield ranges

When matching to a pattern, you can specify a range of words.

The format is as follows:

```
(beg:end)
```

Examples are:

(1:3)	Specifies a range of words 1 - 3
(–3:–1)	Specifies a range of the third word from the last to the last word
(1:–1)	Specifies a range of the first word to the last word (note that by using ? for the last word makes action more efficient)

If you have the address 123 - A B Main St, you can use the following pattern:

```
^ | - | (1:2)
COPY [3] {HouseNumberSuffix}
RETYPE [2] 0
RETYPE [3] 0
```

This pattern results in A and B being moved to the {HouseNumberSuffix} (house number suffix) field. This pattern also retypes A and B to NULL tokens (and similarly retyping the hyphen) to remove them from consideration by any further patterns in the file.

Applying the universal class

You can combine the universal (**) class with other operands to restrict the tokens grabbed by the class. The universal class can be null, signifying no tokens.

The class ** matches all tokens. For example, if you use a pattern of **, you match 123 MAIN ST and 123 MAIN ST, LOS ANGELES, CA 90016, and so on. The following pattern matches to all tokens before the type (which can be no tokens) and the type:

```
** | T
```

Thus, 123 N MAPLE AVE matches with operand [1] being 123 N MAPLE and operand [2] being AVE.

The universal class can be null. No tokens are required to precede the type. AVENUE also matches this pattern with operand [1] being **NULL**.

In the following pattern, the ** refers to all tokens between the numeric and the street type:

```
^ | ** | T
```

In the example, the class ^ (caret) and the type (T) class define the start and end of the ** class. The class ** can contain numbers that are in addition to the class ^ but not any additional street type tokens.

You can specify a range of tokens for an ** operand. For example, the following pattern matches a numeric followed by at least two nonstreet-type tokens followed by a street type:

```
^ | ** (1:2) | T
```

Operand [2] consists of exactly two nonstreet-type tokens. This matches 123 CHERRY TREE DR, but not 123 ELM DR. Only one token follows the number. You can specify ranges from a single token, such as (1:1), to all the tokens, such as (1:–1).

The pattern **(1:1) results in much slower processing time than the equivalent & to match to any single token. However, you do not want to use & in a pattern with **, such as ** | &, because the first token encountered is used by &. Value checks or appropriate conditions that are applied by using & with ** can make sense. For example:

```
** | & = "123", "ABC"
```

No conditional values or expressions are permitted for operands with **.

Using the end of field specifier ($)

The $ specifier does not match any real token, but denotes the end of the pattern.

A pattern condition without the $ specifier can represent a portion of the field, such as the city, state, and postal code information in a U.S. address. For example, Littleton, MA 01460, and LITTLETON MA 01460-6245 match the following pattern:

```
? | S | ^
```

However, the hyphen (-) and the ZIP+4, 01460-6245, are not part of the match. To include the postal code, 01460-6245, as part of the match condition, use the pattern as follows:

```
? | S | ^ | - | ^ | $
```

Any input data following the postal code is not part of the match.

Using floating positioning specifier

You use positioning specifiers to modify the placement of the pattern matching.

For the patterns documented so far, the pattern had to match the first token in the field. For example, the following pattern matches MAPLE AVE and CHERRY HILL RD, but does not match 123 MAPLE AVE, because a number is the first token:

```
? | T
```

You can use floating specifiers to scan the input field for a particular pattern. The asterisk (*) is a positioning specifier and means that the pattern is searched, from left to right, until there is a match or the entire pattern is scanned. You can use the asterisk (*) to indicate that the class immediately following is a floating class.

If you have apartment numbers in the address, to simplify your data, you might want to scan for, process, and retype the apartment numbers to NULL so that basic patterns can process the core address. For example, addresses such as `123 MAIN ST APT 34` and 770 KING ST FL 3 RM 101 contain a basic street address with additional information. The following pattern searches for the unit and floor information, populates the appropriate dictionary fields, and removes the unit and floor information from further processing. U is the class for unit and F is the class for floor.

```
*U | ^
COPY_A [1] {UnitType}
COPY [2] {UnitValue}
RETYPE [1] 0
RETYPE [2] 0


*F | ^
COPY_A [1] {FloorType}
COPY [2] {FloorValue}
RETYPE [1] 0
RETYPE [2] 0
```

Retyping the tokens to **NULL** removes the tokens from consideration by any patterns later in the Pattern-Action file. You prevent recounting all combinations of possibilities. Now, the data to be processed is 123 MAIN ST and 770 KING ST. Both entries have the pattern: ^ | ? | T.

Processing a portion of the data by using the floating specifier simplifies the two input fields and makes the input pattern the same. The standardization task is made easier.

Floating positioning specifiers operate by scanning a token until a match is found. If all operands match, the pattern matches. If the operands do not match, the scanner advances one token to the right and repeats the process. This is like moving a template across the input string. If the template matches, the process is done. Otherwise, the template advances to the next token.

Note: There can only be one match to a pattern in an input string. After the actions are processed, control goes to the next pattern, even though there might be other matches on the line.

The asterisk must be followed by a class. For example, the following operands are valid with a floating positioning specifier followed by a standard class:

```
* U
* ?
* ^
```

There can be more than one floating positioning specifier in a pattern. For example, the following operands match to JOHN DOE 123 CHERRY HILL NORTH RD:

```
*^ | ? | *T
```

Operand [1] is 123. Operand [2] is CHERRY HILL. Operand [3] is RD. NORTH is classified as a directional (D) so it is not included in the unknown string (?).

Using the reverse floating positioning specifier

The reverse floating positioning specifier, indicated by a number sign (#), is similar to the floating positioning specifier (*) except that scanning proceeds from right to left instead of from left to right.

You can use this specifier to search for items that appear at the end of a field, such as postal code, state, and apartment designations.

The reverse floating positioning specifier must only appear in the first operand of a pattern, since it is used to position the pattern. For example, if you wanted to find a postal code and you have given the state name the class S, the following pattern scans from right to left for a state name followed by a number:

```
#S | ^
```

If you have an input string CALIFORNIA 45 PRODUCTS, PHOENIX ARIZONA 12345 DEPT 45, the right to left scan positions the pattern to the ARIZONA. The number following causes a match to the pattern.

If no match is found, scanning continues to the left until a state followed by a number is found. If you are limited to the standard left-right floating positioning specifier (*S | ^), the CALIFORNIA 45 is incorrectly interpreted as a state name and postal code.

Using the fixed position specifier

The fixed position specifier is positioned at a particular operand in the input string.

Sometimes it is necessary to position the pattern matching at a particular operand in the input string. This is handled by the %*n* fixed position specifier. Examples of the fixed position specifier are:

%1	Matches to the first token
%2	Matches to the second token
%–1	Matches to the last token
%–2	Matches to the second from last token

The positions can be qualified by following the %*n* with a token type. Some examples are:

%2^	Matches to the second numeric token
%-1^	Matches to the last numeric token
%3T	Matches to the third street type token
%2?	Matches to the second set of two or more unknown alphabetic tokens

You can use the fixed position specifier (%) in only two ways:

- As the first operand of a pattern
- As the first and second operands of a pattern

The following pattern is allowed and matches the second numeric token as operand [1] and the third leading alpha token that follows as operand [2]:

```
%2^ | %3<
```

The fixed position specifier treats each token according to its class. The following examples illustrate how to use the fixed position specifier for the input field:

```
John Doe
123 Martin Luther St
Salt Lake
```

%1 1	Matches to the first word in the first string: JOHN
%1 2	Matches to the second word in the first string: DOE
%2 ?	Matches to the second string of unknown alphabetic words: MARTIN LUTHER
%2 1	Matches to the first word in the second string: MARTIN
%-2 –1	Matches to the last word in the next to the last string: LUTHER
%3+	Matches to the third single alphabetic word: MARTIN
%–1 ?	Matches to the last string of unknown alphabetic words: SALT LAKE

%–1+	Matches to the last single alphabetic word: LAKE

The position specifier does not continue scanning if a pattern fails to match (unlike * and #).

Assuming the input value S is classified as a D for direction, the following pattern matches the 789 S in the string 123 A 456 B 789 S:

```
%3^ | D
```

That same pattern does not match 123 A 456 B 789 C 124 S because the third number (789) is not followed by a direction.

Negation class qualifier

The exclamation point (!) is used to indicate NOT.

The following pattern language syntax shows how to use the negative to specify matching:

!T	Match to any token except a street type
!?	Match to any unknown token

The following example matches to SUITE 3, APT GROUND but not to SUITE CIRCLE because CIRCLE is classified as a street type (T):

```
*U | !T
```

The phrase RT 123 can be considered to be a street name only if there is no unknown word following, such as RT 123 MAPLE AVE. You can use the following pattern to create a non-match to the unknown word:

```
*T | ^ | !?
```

This pattern matches to RT 123, but not to RT 123 MAPLE AVE because an unknown alphabetic character follows the numeric operand.

You can combine the negation class with the floating class (*) only at the beginning of a pattern. For example, when processing street addresses, you might want to expand ST to SAINT where appropriate.

For example, change 123 ST CHARLES ST to 123 SAINT CHARLES ST, but do not convert 123 MAIN ST REAR APT to 123 MAIN SAINT REAR APT. You can use the following pattern and action set:

```
*!? | S | +
RETYPE [2] ? "SAINT"
```

The previous example requires that no unknown class precede the value ST because tokens with this value have their own class of S.

Conditional patterns

When standardizing addresses, some addresses require that conditions be attached to pattern matching. Providing conditional values in patterns allows correct processing of specific cases.

Simple conditional values

A simple condition is expressed by the typical pattern operand followed by an equal sign and a value.

Alphabetic values must be in quotation marks. A condition to test for the presence of a token with value MAPLE followed by a street type is:

```
*? = "MAPLE" | T
```

The `*? = "MAPLE"` is a token with a condition and is also an operand.

If the word SOUTH is in the classifications, you can test explicitly for SOUTH by using D = "SOUTH" or for any direction with the standard abbreviation S by using D = "S" for the operand.

You enter numeric values without quotes. The following pattern-action set matches to 1000 MAIN but not to 1001 MAIN.

```
* ^ = 1000 | ?
```

The equality operator (=) tests both the standardized abbreviation (if the token is found in the .cls file and has an abbreviation) and the original token value for equality to the operand. Two additional operators are available for testing equality to the abbreviation only or the original token value only. The operators are as follows:

=A=	Only tests the abbreviation from the .cls file
=T=	Only tests the original token value only

For example, to properly handle AVE MARIA LANE, test for equality with the original token value:

```
*T =T= "AVE" | + | T
RETYPE [1] ?
```

As an operand, `*T =T= "AVE"` ensures that AVE is coded and not another value that maps to the same abbreviation leaving input of AVENUE MARIA unchanged. Similarly, use =A= if you only want a test on the abbreviation.

Conditional expressions

The conditional expression is enclosed in brackets immediately following the pattern operand.

If simple values or tables of values are not sufficient to qualify pattern matching, you can use conditional expressions. These expressions have the following format:

operand [*conditional expression*]

A simple conditional expression consists of an operand, a relational operator, and a second operand. The following are the relational operators:

<	Original token is less than
>	Original token is greater than
=	Abbreviation or original token is equal to
=A=	Abbreviation is equal to
=T=	Original token is equal to

<=	Original token is less than or equal to
>=	Original token is greater than or equal to
!=	Abbreviation or original token is not equal to
!=A=	Abbreviation is not equal to
!=T=	Original token is not equal to

The left operand can be any of the following elements:

- A variable name
- The field contents for the current operand
- The contents for any dictionary field

The right operand can be any of the following elements:

- A variable name
- A literal
- A constant

The complete format of a conditional expression is:

left-operand relational-operator right-operand

The following table explains the expression:

Operation	Valid values
left-operand	{} {} PICT {} LEN {*field-name*} {*field-name*} PICT {*field-name*} LEN *variable-name* *variable-name* PICT *variable-name* LEN <*arithmetic-expression*>
relational-operator	< > <= >= != =
right-operand	Literal Constant Variable name

Current operand contents

The contents of the current operand are in left and right braces ({ }).

An example that uses calendar dates best illustrates this concept. When verifying dates, you want to verify the length of numbers. One example is as follows:

```
^ [{}LEN=4] | - | ^ [{}LEN=2] | - | ^ [{}LEN=2]
; format for ccyy-mm-dd
```

This pattern matches on 2009-07-08 but not on 07-08-09.

The pattern operands in the preceding example have the following meaning:

Operand [1]	^	A 4-digit number
Operand [2]	-	A hyphen
Operand [3]	^	A 2-digit number
Operand [4]	-	A hyphen
Operand [5]	^	A 2-digit number

When character literals are in an equality test, the standard abbreviation is tested if one is available. If this fails, the original input is tested. The following examples show pattern operands when you have ROAD RD T in your classifications:

`T [ {} = "RD" ]`	Compares the abbreviation of the current operand (RD) to the literal RD
`T [ {} = "ROAD" ]`	Compares the entire input operand to the literal (because it fails to compare to the abbreviation)
`T [ {} =A= "RD"]`	Compares only the abbreviation of the current operand to RD
`T [ {} =T= "ROAD"]`	Compares the original value of the token to ROAD and not to the abbreviation
`T [ {} <= "RD" ]`	When comparisons (other than the equality operators) are specified, the original input is used rather than the abbreviation. This is true for any comparison to a literal value. If the original value is RD, the result is true, but, if the original value is ROAD, the result is false.

Dictionary field contents

The dictionary field is identified by the field name that is listed in the first column of the table in the dictionary definition file (.DCT).

Sometimes you need to test a value placed in a dictionary field from an earlier process or from a pattern-action set that was previously executed. This can be accomplished by specifying the field name enclosed in braces.

For example, you have two streets named EAST WEST HIGHWAY. In postal codes 20100 - 20300, the name of the street is EAST WEST, and in postal codes 80000 - 90000, WEST is the name of the street and EAST is the direction. If the postal code field is populated by an earlier process and named {ZipCode}, you can use the following pattern-action sets:

```
^ | D = "E" | D = "W" | T = "HWY" | [ {ZipCode} >= 20100 & {ZipCode} <= 20300 ]
COPY [1] {HouseNumber}   ; move house number to {HouseNumber} field
COPY [2] temp   ; concat EAST WEST and move
CONCAT [3] temp   ;   to street name field
COPY temp {StreetName}
COPY_A [4] {StreetSuffixType}   ; move HWY to street type field

^ | D  = "E" | D = "W" | T = "HWY" | [ {ZipCode} >= 80000 & {ZipCode} <= 90000 ]
COPY [1] {HouseNumber}   ; move house number to {HouseNumber} field
COPY_A [2] {StreetPrefixDirectional}   ; move EAST to direction
COPY [3] {StreetName}   ; move WEST to street name
COPY_A [4] {StreetSuffixType}   ; move HWY to street type field
```

In this pattern, the operand [{ZipCode} >= 20100 & {ZipCode} <= 20300] states:

{ZipCode}	The value in the postal code field
>=	is greater than or equal to
20100	the number 20100
&	and
{ZipCode}	the value in the postal code field
<=	is less than or equal to
20300	the number 20300

The logical operator & is used to connect two conditional expressions. The condition is placed in a separate operand. It can also be placed in the operand for HWY; for example:

```
^ | D = "E" | D = "W" | T = "HWY"  [ {ZipCode} >= 80000 & {ZipCode} <= 90000 ]
```

These two forms are identical in function. However, the first form is easier to read because the conditions are placed in separate pattern operands.

When conditions only reference dictionary field contents (and not any pattern operand), as in the preceding example with {ZipCode}, the condition must follow all pattern operands. The following example is not valid since the second operand does not reference an input field and a third operand (T) follows the condition:

```
^ | [{ZipCode} = 80000] | T
```

The correct form is:

```
^ | T | [{ZipCode} = 80000]
```

If you use in a condition a dictionary field name that is not defined in the dictionary definition file, any test on its value always returns FALSE. If you are testing for NULL contents, the test returns TRUE; for example:

```
{ZZ} = ""
```

This facility allows use of the same general Pattern-Action files on projects which dispense with certain fields (as opposed to ending the program with an invalid field error).

Literals: character constants and user variables

Literals are character constants. They are represented by enclosing a string in quotes.

Numeric constants are referenced to by coding a number. Negative numbers and decimal points are not permitted in numeric constants. The following pattern operand matches on a number equal to 10000:

```
^ [{} = 10000]
```

The following example matches to the text MAIN:

```
? [ {} = "MAIN" ]
```

If an unknown operand (?) is specified, multiple words are concatenated to a single word. To match to CHERRY HILL, use the following pattern:

```
? [ {} = "CHERRYHILL" ]
```

A NULL or empty value is indicated by two consecutive quote marks.

```
[ user_variable = ""]
```

You can use any of the relational operators for character strings. The following example matches on all strings starting with MA or greater, including MA, MAIN, NAN, PAT, but not ADAMS, M, or LZ:

```
? [ {} > "MA" ]
```

The equality (=) is tested on the abbreviation for the operand first if one exists, and then on the full operand. The other relational operators test on the full operand and not the abbreviation.

You can define variables to which you assign specific values by using the actions. You can test variables for specific values within conditions. Variables must be named according to the following conventions:

- The first character must be alphabetic.
- The name cannot exceed 32 characters.

After the first alphabetic character, you can use any combination of alphanumeric characters or the underline (_).

For example, if you set the variable *postcode* to the postal code, you can test to see if the post code is 12345 as follows:

```
[postcode = 12345]
```

This type of condition can be a separate pattern operand or combined with a standard class. For example, the following two patterns produce identical results:

```
^ | [postcode = 12345]
^ [postcode = 12345]
```

If a user variable is set to a numeric value, its type is numeric. If it is set to a literal value, its type is character.

If a condition only references variables or dictionary fields, and not current input operands, the condition must follow all operands. The following example is not valid since the second operand does not reference an input field and a third operand follows:

```
^ | [postcode = 12345] | T
```

You must replace it with:

```
^ | T | [postcode = 12345]
```

Referencing lengths of operands (LEN)

LEN represents the length of an operand.

A special LEN function is available. You can use the expression in one of the following three ways:

{} LEN	Length of the current operand
variable LEN	Length of the contents of a variable
{*field-name*} LEN	Length of the contents of a dictionary field

If you want to search for a nine-digit postal code of 12345-6789, check for a five-digit number followed by a hyphen and a four-digit number, for example:

```
^ [ {} LEN = 5] | - | ^ [{} LEN = 4]
```

If the numerics do not match the length, the pattern does not match.

Similarly to test the length of a variable, the following pattern matches if the variable contains five characters:

```
? [ temp LEN = 5]
```

Finally, to test the length of a dictionary field, use the field name within braces:

```
[ {StreetName} LEN = 20 ]
```

Any trailing blanks are ignored in the calculation of the length. Leading blanks are counted. For example, if a user variable or a dictionary field is set to " XY" (two leading spaces), the length of either is 4.

Referencing formats of operands (PICT)

The picture defines how the numeric and alphabetic characters are formatted.

In some cases, you must test for special formats. You can use the PICT (picture) function. For example, Canadian postal codes have the form character-number-character (space) number-character-number; for example, `K1A 3H4`. You can use the PICT function to represent these sequences:

```
@ [{} PICT = "cnc"] | @ [{} PICT = "ncn"]
```

The @ (at sign) matches to a complex type (mixed numeric and alphabetic characters).

cnc	For character-number-character
ncn	For number-character-number

Some British postal codes use the form character-character-number (space) number-character-character; for example, `AB3 5NW`.

```
< [{} PICT = "ccn"] | > [{} PICT = "ncc"]
```

The PICT clause works for dictionary field values:

```
[ {ZipCode} PICT = "ccn" ]
```

The PICT operand works for user variables:

```
[ temp PICT = "ncncn" ]
```

Only the equality (=) and inequality (!=) operators can be used with PICT comparisons.

Referencing substrings of operands

A special form is provided in the patterns to test a portion of an operand.

These portions are called substrings. The following forms are valid:

{} (beg:end)	Substring of the current operand
variable (beg:end)	Substring of the content of the user variable

field-name (beg:end)	Substring of the content of the dictionary field

The (beg:end) specifies the beginning and ending character to extract. The first character in the string is 1, the last character is –1, and so on.

For example, German style addresses have the street type appended to the end of the street name. Thus, HESSESTRASSE means HESSE STREET. The substring form can be used to test for these suffixes. Consider an input address of HESSESTRASSE 15. The following pattern matches to all words ending in STRASSE that are followed by a numeric:

```
+ [{} (-7:-1) = "STRASSE"]  | ^
```

Similarly, variables and fields can be tested:

```
[temp(2:4) = "BCD"]
[{StreetName}(1:4) = "FORT"]
```

When conducting substring tests on multi-token values (?), remember that separating spaces are removed. To test MARTIN LUTHER KING, specify the pattern as follows:

```
? [{} (1:12) = "MARTINLUTHERKING"]
```

Series of conditional values

You specify a series of conditional values by delimiting the entries with either spaces or commas.

The following examples are equivalent and mean a street type whose value or standardized abbreviation is either RD or AV or PL:

```
T = "RD", "AV", "PL"
T = "RD" "AV" "PL"
```

Numeric series can be represented in the same manner, except without quotation marks. Optionally, you can use the abbreviation equality operator =A= or the original value operator =T=, such as:

```
T =A= "RD", "AV", "PL"
T =T= "RD", "AV", "PL"
```

A series of values can be tested against a dictionary field in a similar fashion:

```
^ | T | [ {StreetName} = "MAIN", "ELM", "COLLEGE" ]
```

The following pattern tests country code (specified as CC) to determine the processing to be used:

```
; Armenia, Azerbaijan, China, Georgia, Russia, Tajikstan
;
[ {CC}="ARM" ,"AZE" ,"CHN" ,"GEO" ,"RUS" ,"TJK" ]
CALL Area_Format_B
```

In the case where you are testing the dictionary field value instead of a normal pattern operand, the test must follow all pattern operands including end-of-field.

```
^ | ? | T | $ | [ {StreetName} = "MAIN", "ELM", "COLLEGE" ]
```

The test [{StreetName} = "MAIN", "ELM", "COLLEGE"] follows all pattern operands, including the end of field specifier ($).

Tables of conditional values

You can specify many conditional values by creating a table of values file.

You can use a table of values files when you have many values that would be difficult to include in a conditional statement, or when you prefer to update values in a table rather than modifying the pattern-action file.

Tables can be specified as follows:

```
@TABLE_FILE_NAME.TBL
```

For example, you want to test a number to see if it is one of a series of postal codes. First, edit the pattern-action file to reference the file that list postal codes. A placeholder is created for the file in the InfoSphere DataStage® and QualityStage Designer. Next you edit the file that list postal codes. Ensure that you have one line for each postal code. As an illustration, this file is named POSTCODE.TBL and looks as follows:

```
90016
90034
90072
90043
...
```

A pattern matching city, state, and postal code might look like:

```
? | S | ^ = @POSTCODE.TBL
```

This pattern is intended to match to cases with city name, state, and post code. If the numeric operand is in the list, the pattern matches; otherwise it does not. LOS ANGELES CA 90016 matches, but CHICAGO IL 12345 does not because the postal code is not in the table.

The table file name can contain complete or relative path information, including an environment variable. If a rule set has a path specified on a command line, that path is assumed for value files used in that Pattern-Action file for the rule set and the path cannot be specified a second time.

The dictionary field contents can also be tested against tables of values:

```
^ | T | {StreetName} = @STRTNAME.TBL
```

If the dictionary field contents are not pattern operands, the test against a table of values must follow all pattern operands, including an end-of-field operand. The following example is not valid, since a pattern operand follows the table test:

```
^ | {StreetName} = @STRTNAME.TBL | T
```

Using arithmetic expressions

You can include arithmetic expressions as the left operand of a conditional expression.

During standardization, rule sets only test the output of arithmetic expressions. The rule sets do not generate the answer to arithmetic expressions to populate the output.

The available arithmetic operators are:

+	Addition

–	Subtraction
*	Multiplication
/	Division
%	Modulus

Arithmetic is limited to one operation per expression. Parentheses are not permitted. The modulus operation is the remainder of an integer division. For example, x % 2 is zero if the number is divisible by two. It is one if the number is odd.

An arithmetic expression is

left-arithmetic-operand arithmetic-operator right-arithmetic-operand

Operation	**Valid Value**
left-arithmetic-operand	variable-name {*field-name*} {}
arithmetic-operator	+ – * / %
right-arithmetic-operand	variable-name constant

Examples of arithmetic expressions are:

`temp –2`	The value of temp –2
`{} % 2`	The current operand value of modulo 2

The following conditional expression can be used for matching to even-numbered houses.

```
^ [{} % 2 = 0]
```

Even numbers are divisible by two, thus the house number modulo two is zero. The arithmetic expression appears on the left side of the relational operator (the equal sign).

The following syntax is a conditional expression to see if the current operand divided by three is greater than the contents of variable temp:

```
^ [{} / 3 > temp]
```

Again, note that the field references and the arithmetic expression are to the left of the relational operator. Other examples are:

```
[ temp * temp2 > temp3 ]
[ {ZipCode} + 4 > 12345]
```

Combining conditional expressions

You can combine conditional expressions by using logical operators.

&	AND
|	OR

Note: The OR operator is inclusive, if part of the statement is TRUE, the result is TRUE.

To test for even-numbered houses greater than 1000:

```
^ [{} % 2 = 0 & {} > 1000]
```

To test for houses in the range of 1000 to 10000:

```
^ [{} >= 1000 & {} <= 10000]
```

To test for houses less than 100 or greater than 1000:

```
^[{} < 100 | {} > 1000]
```

To test for even-numbered houses, and for half of the value in temp to be greater than 50, and with a postal code greater than 12345:

```
^ [{} % 2 = 0 & temp / 2 > 50 & {ZipCode} > 12345]
```

Parentheses are not permitted. All operations are executed left to right. Within a single bracket-delimited condition, AND and OR operators cannot be mixed. An error message is printed if such a case is encountered. Operator precedence can be obtained by using separate pattern operands if possible. The arithmetic expression ((a | b) & (c | d)) can be represented by:

```
[a | b] | [c | d]
```

The vertical lines (`|`) within the brackets are logical OR operations, and vertical lines outside the brackets are operand separators. In the previous example, `a`, `b`, `c`, and `d` represent conditional expressions.

Action statements

Action statements execute the standardization rules which are the actions associated with the pattern of tokens in the Pattern-Action file.

Any action referring to a field that is not defined in the dictionary definition file is ignored, and a warning message is added to the IBM InfoSphere DataStage and QualityStage Director log.

Copying information

The COPY action copies information from a source to a target.

The format is:

```
COPY source target
```

The *source* can be any of the following:

Type	Description
operand	A pattern operand ([1], [2], ...)

Type	Description
substring operand	A substring of the pattern operand
mixed operand	Leading numeric or character subset
user variable	A user-defined variable
field name	A key reference ({StreetName}, ...)
literal	A string literal in quotes ("SAINT")
constant	A numeric value

The *target* can be:

Type	Description
field name	A dictionary field ({StreetName}, ...)
user variable	A user-defined variable

For example, a United States address pattern that matches to 123 N MAPLE AVE is:

```
^ | D | + | T
```

This is accomplished by the following pattern action set:

```
^ | D | + | T
COPY [1] {HouseNumber}
COPY [2] {StreetPrefixDirectional}
COPY [3] {StreetName}
COPY [4] {StreetSuffixType}
EXIT
```

The following operations occur:

- The number operand [1] moves to the house number field {HouseNumber}
- The class D operand (direction) moves to the prefix direction field {StreetPrefixDirectional}
- The unknown alphabetic operand moves to the street name field {StreetName}
- The street type (class T) to the {StreetSuffixType} field.

Copying substrings

A substring of an operand is copied by using the substring operand form.

The simplest form of the COPY action is copying an operand to a dictionary field value. For example:

```
COPY [2] {StreetName}
```

The substring operand only operates on standard operands or user variables. The form is:

```
COPY source(b:e) target
```

The `b` is the beginning column of the string and the `e` is the ending column. The following example copies the first character (1:1) of operand 2 to the street name field:

```
COPY [2](1:1) {StreetName}
```

The following example copies the second through fourth characters of the contents of the variable temp to the {StreetName} field:

```
COPY temp(2:4) {StreetName}
```

You can use a negative one (–1) to indicate the last character. A negative two (–2) indicates the next to last character, and so on. The following example copies the last three characters of operand 2 to the street name field:

```
COPY [2](-3:-1) {StreetName}
```

Copying leading and trailing characters

When handling leading alpha, leading numeric, or mixed class tokens, you can isolate the substrings based on the character type.

The four possible mixed operand specifiers are:

(n)	All leading numeric characters
(–n)	All trailing numeric characters
(c)	All leading alphabetic characters
(–c)	All trailing alphabetic characters

These specifiers can be used for standard operands or for user variables.

For example, with the address 123A MAPLE AVE, you want the numbers 123 to be recognized as the house number and the letter A to be recognized as a house number suffix. You can accomplish this with the following pattern:

```
> | ? | T
COPY [1](n) {HouseNumber}
COPY [1](-c) {HouseNumberSuffix}
COPY [2] {StreetName}
COPY_A [3] {StreetSuffixType}
EXIT
```

Note that the first operand > is the appropriate class (leading numeric). These leading and trailing specifiers are mostly used with the **>** OR **<** operands. However, the trailing and leading specifiers can be used to separate user variables too. In the following example, XYZ is copied to StreetName:

```
COPY "456XYZ" tmp2
COPY tmp2(-c) {StreetName}
```

Copying user variables

The type of a target user variable is determined by the type of the source.

A user variable can be the target and the source of a **COPY**. The following are some examples:

Copy samples	Description
`COPY [1] temp`	Operand 1 is copied to a variable named *temp*.
`COPY "SAINT" temp`	The literal "SAINT" is copied to the variable *temp*.
`COPY temp1 temp2`	The contents of variable *temp1* is copied to *temp2*.
`COPY temp1(1:3) temp2`	The first three characters of *temp1* are copied to *temp2*.

User variables can consist of 1 to 32 characters where the first character is alphabetic and the other characters are alphabetic, numeric, or an underscore (_) character.

Copying dictionary columns

Dictionary columns can be copied to other dictionary columns or to user variables.

The following example shows dictionary columns being copied to other dictionary columns or user variables:

```
COPY {HouseNumber} {HC}
COPY {HouseNumber} temp
```

Copying standardized abbreviations

The **COPY_A** action copies the standardized abbreviation for an operand to a target. Whereas, the **COPY** action copies the input to a target.

Standardized abbreviations are coded for entries in the classifications for the rule set. They are not available for default classes, such as a number, an alphabetic unknown, and so on.

You can use the **COPY_A** action to copy the abbreviation of an operand to either the dictionary column or a user variable. The following shows an example:

```
^ | ? | T
COPY [1] {HouseNumber}
COPY [2] {StreetName}
COPY_A [3] {StreetSuffixType}
```

The third line copies the abbreviation of operand three to the street type column. Similarly, the following example copies the standard abbreviation of operand three to the variable named temp:

```
COPY_A [3] temp
```

Abbreviations are limited to a maximum of 25 characters.

The **COPY_A** action copies the standardized abbreviation to the dictionary field rather than the original token. **COPY_A** can include a substring range, in which case the substring refers to the standard abbreviation and not the original token, as in **COPY**.

Copying with spaces

You can preserve spaces between words by using the **COPY_S** action.

When you use COPY to copy an alphabetic operand (?) or a range of tokens (**) to a dictionary column or to a user variable, the individual words are concatenated together.

COPY_S requires an operand as the source and either a dictionary column or a user variable as a target. For example, with the following input string:

```
123 OLD CHERRY HILL RD
```

A standard **COPY** produces OLDCHERRYHILL, but in the following pattern, **COPY_S** can be used as shown here:

```
^ | ? | T
COPY [1] {HouseNumber}
COPY_S [2] {StreetName}
COPY_A [3] {StreetSuffixType}
```

The {StreetName} column contains: OLD CHERRY HILL.

If you use the universal matching operand, all tokens in the specified range are copied. For example, consider removing parenthetical comments to a column named {AdditionalInformation}. If you have the following input address:

```
123 MAIN ST (CORNER OF 5TH ST) APARTMENT 6
```

You can use the following pattern to move CORNER OF 5TH ST to the column {AdditionalInformation}. The second action moves the same information to the user variable temp:

```
\( | ** | \)
COPY_S [2] {AdditionalInformation}
COPY_S [2] temp
```

Only when copying the contents of an operand does **COPY** remove spaces; thus, the restriction that the source for a **COPY_S** action can only be an operand. For all other sources (literals, formatted columns, and user variables), **COPY** preserves spaces.

Copying the closest token

The **COPY_C** action copies corrected input values as matched to entries in the classifications (.CLS). When a token has an uncertainty threshhold and is in the classifications, that you can copy a corrected version of the input rather than the abbreviation value.

When matching under uncertainty to entries in the classifications, you might want to use **COPY_C** action so that the complete token is spelled correctly rather than copying an abbreviation.

For example, if you have state name table with an entry such as:

```
MASSACHUSETTS   MA   S   800.0
```

If Massachusetts is misspelled on an input record (for example, Masssachusettts), you want to copy the correct spelling to the dictionary column. The following action places the full correctly spelled token MASSACHUSETTS in the proper column:

```
COPY_C [operand-number] {column name}
```

For COPY_C, source can only be an operand because COPY_C uses the closest token from classifications.

Copying initials

The **COPY_I** action copies the initial character (first character of the relevant tokens) from a source to the dictionary column rather than the entire value.

The dictionary column can be a dictionary field or a user variable. You can use COPY_I within a pattern action set or as a **POST** action.

The value MAPLE puts the M into {NameAcronym}, if you use **COPY_I** in the following manner:

```
?
COPY_I [1] {NameAcronym}
```

For a multi-token alphabetic string such as John Henry Smith, the output value depends on the target. If the target is a user variable, the output value from the **COPY_I** action is JHS.

If you use **COPY_I** as a **POST** action, the source must be a dictionary column and the target must be a dictionary column. You generally use **COPY_I** to facilitate array matching.

For example, the company name INTERNATIONAL BUSINESS MACHINES is distributed into dictionary columns C1 through C5 (such that C1 contains INTERNATIONAL, C2 BUSINESS, C3 MACHINES, and C4 and C5 are blank). The following set of **POST** actions put the value IBM into the company initials dictionary column CI.

```
\POST_START
COPY_I {C1} {CI}
CONCAT_I {C2} {CI}
CONCAT_I {C3} {CI}
CONCAT_I {C4} {CI}
CONCAT_I {C5} {CI}
\POST_END
```

When used as a POST action, **COPY_I** takes only the first character from the source column. For example, if, in the previous sample, C1 INTERNATIONAL DIVISION, the result is still IBM.

Referencing dictionary fields from another rule set

You can use the dictionary fields of one rule set in another rule set. You might want to reference other rule sets to create a test case that helps you determine which patterns to use.

The reference syntax is as follows: {*<DictionaryFieldName>* OF *<RuleSetName>*}.

Within the Standardize stage, you can check the conditions and actions from dictionary fields of a previously processed rule set and determine if a different set of actions is required for the same pattern within the current rule set. You must ensure that both the referenced rule set and referencing rule set are specified in the same Standardize stage. Ensure that the referenced rule set precedes the referencing rule set in the stage properties list of rule sets. For example, if USADDR references USAREA, then the USAREA rule set must proceed USADDR in the list of rule sets in the Properties window of the Standardize stage.

The two rule sets domains, area and address, can provide more examples. Address is processed independently of the area. In this example, you check to see if the CAAREA rule set generates Quebec as the associated province. If Quebec is the Canadian province then you want CAADDR to handle the address differently than the way address patterns are handled for other Canadian provinces.

You reference the CAAREA dictionary fields in the CAADDR rule set. In the Standardize stage, you ensure that CAAREA is defined and listed before CAADDR.

In CAADDR.PAT file, the pattern checks to see the contents of the dictionary field, `ProvinceAbbreviation of CAAREA` is `QC`. The following patterns show how you can create the reference in CAADDR:

```
^ | T | ? | $  [ {ProvinceAbbreviation OF CAAREA} ="QC" ]
COPY [1] {CivicNumber}
COPY_A [2] {StreetSuffixType}
COPY_S [3] {StreetName}
```

```
RETYPE [1] 0
RETYPE [2] 0
RETYPE [3] 0
RETURN
```

The preceding pattern specifies that, in the province of Quebec, the street type (as defined in CAADDR.CLS and represented by T), moves to StreetSuffixType, for ease of matching.

Then, for all other Canadian province, the street type moves to StreetPrefixType.

```
^ | T | ? | $
COPY [1] {CivicNumber}
COPY_A [2] {StreetPrefixType}
COPY_S [3] {StreetName}
RETYPE [1] 0
RETYPE [2] 0
RETYPE [3] 0
RETURN
```

A Canadian street address might be: 498 Rue Bordeaux. The output from the rule sets would be as follows:

CAAREA	CivicNumber	StreetPrefixType	StreetName	StreetSuffixType
Quebec	498		Bordeaux	Rue
Other Canadian provinces	498	Rue	Bordeaux	

Moving information

You can use **MOVE** to move a user variable or a dictionary column to a dictionary column.

The **MOVE** action copies information, from source to target, and then erases the value of the source. The only valid sources for the **MOVE** action is a user variable or a dictionary column. You cannot use the MOVE action with operand. Examples of the **MOVE** action are as follows:

```
MOVE Country_Code {ISOCountryCode}
```

and

```
MOVE {HouseNumber} {HouseNumberSuffix}
```

Concatenating information

The **CONCAT** action and the **PREFIX** action provide two actions for concatenating information in a rule set.

CONCAT action

CONCAT concatenates information to a user variable or a dictionary column. The source can be an operand, literal, or user variable. For example, fractions, such as ½ and ¼, can be copied to a single column in the dictionary by using the following pattern action set:

```
^ | / | ^
COPY [1] temp
CONCAT [2] temp
CONCAT [3] temp
COPY temp {Fractions}
```

The {Fractions} column contains the entire fraction (½).

User variables are most often used with the **CONCAT** action to form one field. If you want to copy two directions with spaces (for example: EAST WEST) into a single dictionary column, you create the following actions:

```
D =T= "EAST"| D =T= "WEST" | T
COPY [1] temp
CONCAT " " temp
CONCAT [2] temp
COPY temp {StreetName}
```

This pattern tests for the specific directions EAST and WEST. The first operand is copied to the user variable temp. The contents of temp is now EAST. The next line concatenates a space to the variable temp. The second **CONCAT** appends WEST to the variable temp. The variable now contains EAST WEST. Then, the contents of temp are copied to the street name field.

Note: A literal with a single space is concatenated to the variable. The same results cannot be obtained by concatenating directly into the dictionary column. Dictionary columns are null at the time of initialization. Adding a space does not change the contents of the column and does not alter future actions.

CONCAT permits substring ranges. For example:

```
CONCAT [1](3:-2) {StreetName}
```

From position 3 to the second to last position of the first operand is concatenated to the street name column of the dictionary column.

CONCAT_A concatenates the standard abbreviation instead of the original token data. The source can only be an operand. **CONCAT_A** allows substring ranges. However, the substring refers to the standard abbreviation and not the original token.

CONCAT_I concatenates the initials instead of the original token data. You can use **CONCAT_I** as a **POST** action where the source must be a dictionary column and the target must be a dictionary column.

CONCAT_I, when not used as a **POST** action, allows substring ranges with the substring referring to the initials and not the original token. In most cases, there is a single initial, but for multi-token strings, such as John Henry Smith, the initials are JHS, and substring ranges other than (1:1) make sense.

Learn more the CONCAT action and spaces: When the source is a user variable, the CONCAT action preserves the spaces within the token. When the source is an operand, the CONCAT action does not preserve spaces within the token.

CONCAT does not provide the ability to preserve spaces between tokens matching up to either ? or ** in a pattern statement (such as the **COPY_S** action). To preserve spaces between tokens, you must use the **COPY_S** action to copy the tokens to a user variable and prefixing or concatenating that user variable. To pick up attention text in an input line, refer to the following examples:

Table 2. Examples of how to copy tokens into user variables

Lines within one pattern action set	Description
+="SEE" | **	; grab "SEE JOHN DOE"

Table 2. Examples of how to copy tokens into user variables (continued)

Lines within one pattern action set	Description
COPY [1] temp	`; put "SEE" in user variable temp`
CONCAT " " temp	`; add a space to the end of user variable temp`
COPY_S [2] temp2	`; put "JOHN DOE" (including space) in temp2`
CONCAT temp2 temp	`; concat temp2 onto temp`
COPY temp {AdditionalNameInformation}	`; put "SEE JOHN DOE" into column AdditionalNameInformation`

PREFIX action

The **PREFIX** action adds data to the beginning of a string. The source for **PREFIX** can be an operand, a literal, or a user variable. The target can be a user variable or a dictionary column.

```
COPY "CHARLES" temp
PREFIX "SAINT" temp
```

In the preceding example, the variable *temp* contains SAINTCHARLES.

PREFIX permits substring ranges. For example in the following sample:

```
PREFIX [1](3:-2) {StreetName}
```

From position 3 to the second to last position of the first operand is prefixed to the street name column.

PREFIX_A prefixes the standard abbreviation instead of the original token data. The source must be an operand. **PREFIX_A** allows substring ranges; however, the substring refers to the standard abbreviation and not the original token.

PREFIX_I prefixes the initials instead of the original token data. You can use **PREFIX_I** as a **POST** action where the source must be a dictionary column and the target must be a dictionary column.

PREFIX_I, when not used as a **POST** action, allows substring ranges with the substring referring to the initials and not the original token. In most cases, there is a single initial, but for multi-token strings, such as John Henry Smith, the initials are JHS, and substring ranges other than (1:1) make sense.

Learn more the PREFIX action and spaces: When the source is a user variable, the PREFIX action preserves the spaces within the token. When the source is an operand, the PREFIX action does not preserve spaces within the token.

PREFIX does not let you preserve spaces between tokens matching up to either ? or ** in a pattern statement such as the **COPY_S** action. To preserve spaces between tokens, you must use **COPY_S** to copy the tokens to a user variable and the prefix or concatenate that user variable. Refer to the preceding pattern-action set example.

Converting information

You use the CONVERT action to convert data according to a lookup table or a literal you supply.

You can perform the following actions:

CONVERT
Changes tokens.

CONVERT_S
Concatenates a suffix onto specified tokens.

CONVERT_P
Concatenates the first prefix in a token that matches a value in a lookup table onto specified tokens.

CONVERT_PL
Concatenates the longest prefix in a token that matches a value in a lookup table onto specified tokens.

CONVERT_R
Runs tokens thru the tokenization process again when you implement changes to the tokens.

You can use the following actions to convert characters from languages that are not Latin-based:

TRANS_KH
Converts Katakana characters to Hiragana characters.

TRANS_HK
Converts Hiragana characters to Katakana characters.

TRANS_WN
Converts full-width characters to half-width characters.

TRANS_NW
Converts half-width characters to full-width characters.

Converting place codes

You use conversion with input records that use numeric codes for place names.

The codes are converted to actual place names. You must first create a table file with two columns. The first column is the input value and the second column is the replacement value. For example, the file CODES.TBL contains:

```
001 "SILVER SPRING"
002 BURTONSVILLE   800.0
003 LAUREL
...
```

Multiple words must be enclosed in quotation marks (""). Optional weights can follow the second operand in the previous example to indicate that uncertainty comparisons might be used. The string comparison routine is used on BURTONSVILLE, and any score of 800 or greater is acceptable. The following pattern converts tokens according to the preceding table:

```
&
CONVERT [1] @CODES.TBL TKN
```

The tokens remain converted for all patterns that follow, as if the code is permanently changed to the text.

Convert files must not contain duplicate input values (first token or first set of tokens enclosed in quotes). If duplicate entries are detected, the Standardize stage issues an error message and stops.

Temporary conversion

With the **CONVERT** action, you can specify TEMP to apply a conversion to only the current set of actions.

The TEMP mode is a temporary conversion. The following example converts the suffix of the first operand according to the entries in the table SUFFIX.TBL.

```
CONVERT_S [1] @SUFFIX.TBL TEMP
```

If you have an operand value of HESSESTRASSE and a table entry in SUFFIX.TBL of:

STRASSE	STRASSE	800.0

Operand [1] is replaced with the value:

```
HESSE STRASSE
```

There is now a space between the words. Subsequent actions in this pattern-action set operate as expected. For example, COPY_S copies the two words HESSE STRASSE to the target. COPY, CONCAT, and PREFIX copy the string without spaces. For example, if the table entry is:

STRASSE	STR	800.0

The result of the conversion is HESSE STR. **COPY_S** preserves both words, but **COPY** copies HESSESTR as one word. The source of a CONVERT_P, CONVERT_PL, or CONVERT_S action can be an operand (as in the example), a dictionary field, or a user variable with equivalent results.

Permanent conversion

The mode of TKN provides permanent conversion of a token.

Generally, when you are making a permanent conversion, you specify a retype argument that applies to the suffix with **CONVERT_S** or the prefix with **CONVERT_P** or **CONVERT_PL**. For example, assume that you are using the following **CONVERT_S** statement:

```
CONVERT_S [1] @SUFFIX.TBL TKN T
```

You also have an operand value of HESSESTRASSE and a table entry in SUFFIX.TBL of:

STRASSE	STRASSE	800.0

HESSE retains the class ? because you did not specify a fifth argument to retype the body or root of the word, and STRASSE is given the type for the suffix, such as T, for street type. To perform further actions on these two tokens, you need a pattern of:

```
? | T
```

If no retype class is given, both tokens retain the original class ?.

You might want to retype both the prefix or suffix and the body. When checking for dropped spaces, a token such as APT234 can occur. In this case, the token has been found with a class of < (leading alphabetic character) and an optional fourth argument can retype the prefix APT to U for multiunit and an optional fifth argument can retype the body 234 to ^ for numeric. In the following example, the PREFIX.TBL table contains an APT entry:

```
CONVERT_P [1] @PREFIX.TBL TKN U ^
```

If you want to retype just the body, you must specify a dummy fourth argument that repeats the original class.

Converting multi-token operands

If you are converting multi-token operands that matched to patterns ** or ?, the format of the convert table depends on whether the third argument to CONVERT is TKN or TEMP.

If the third argument is TKN, each token is separately converted.

Thus, to convert SOLANO BEACH to MALIBU SHORES, the convert table must have the following two lines:

SOLANO	MALIBU
BEACH	SHORES

This might produce unwanted side effects, since any occurrence of SOLANO is converted to MALIBU and any occurrence of BEACH is converted to SHORES.

To avoid this situation, the *TEMP* option for **CONVERT** must be used. The combined tokens are treated as a single string with no spaces. Thus, SOLANOBEACH becomes the representation for a ? pattern containing the tokens SOLANO and BEACH. The following entry in the **CONVERT** table accomplishes the proper change:

SOLANOBEACH	"MALIBU SHORES"

In this convert table there must be no spaces separating the original concatenated tokens. When copying the converted value to a dictionary field, **COPY** does not preserve spaces. Therefore, use **COPY_S** if you need to keep the spaces.

Assigning fixed values

You can use **CONVERT** to assign a fixed value to an operand or dictionary column.

This is accomplished by:

```
CONVERT operand literal TEMP | TKN
CONVERT dictionary-field literal
```

For example, to assign a city name of LOS ANGELES to a dictionary column, you can use either of the following actions:

```
COPY "LOS ANGELES" {CT}
CONVERT {CT} "LOS ANGELES"
```

More important, it can be used to convert an operand to a fixed value:

```
CONVERT [1] "LOS ANGELES" TKN
```

TKN makes the change permanent for all actions involving this record, and TEMP makes the change temporary for the current set of actions.

An optional class can follow the TKN. Since converting to a literal value is always successful, the retyping always takes place.

Converting prefixes and suffixes

When a single token can be composed of two distinct entities, a **CONVERT**-type action can be used to separate and standardize both parts.

An example of this is in German addresses, where the suffix STRASSE can be concatenated onto the proper name of the street, such as HESSESTRASSE.

If a list of American addresses has a significant error rate, you might need to check for occurrences of dropped spaces such as in MAINSTREET. To handle cases such as these, you can use the **CONVERT_P** or **CONVERT_PL** action to examine the token for a prefix and **CONVERT_S** for a suffix.

Like **CONVERT_P**, the **CONVERT_PL** action examines the token for a prefix. However, **CONVERT_P** takes the first prefix that matches a value in the lookup table and **CONVERT_PL** takes the longest prefix that matches.

For example, assume that a lookup table contains entries for NORTH and NORTHWEST. For the token NORTHWESTPOINT, the **CONVERT_P** action takes the prefix NORTH and the **CONVERT_PL** action takes the prefix NORTHWEST.

CONVERT_P, **CONVERT_PL**, and **CONVERT_S** use almost the same syntax as **CONVERT**. The first difference is that you must use a lookup table with these actions. The second difference is that you have an optional fifth argument.

```
CONVERT_P source @table_name TKN | TEMP retype1 retype2
CONVERT_PL source @table_name TKN | TEMP retype1 retype2
CONVERT_S source @table_name TKN | TEMP retype1 retype2
```

Argument	Description
source	Can be either an operand, a dictionary field, or a user variable.
retype1	Refers to the token class that you want assigned to the prefix with a **CONVERT_P** or **CONVERT_PL** action or the suffix with a **CONVERT_S**. This argument is optional.
retype2	Refers to the token class that you assigned to the remainder of the token after the conversion, if the source is an operand.

Converting with retokenization

You can use the **CONVERT_R** action to force the new tokens through the tokenization process so that classes and abbreviations are correct.

Use the **CONVERT_R** action when you want to convert a single token into two or more tokens, some of which can be of classes different from the original class.

The syntax for **CONVERT_R** is simpler than the syntax for **CONVERT** since an operand is the target of the action and the argument of TKN is assumed. Use a convert table and the tokenization process retypes automatically:

```
CONVERT_R source @table_name
```

With **CONVERT_R**, the source is the operand.

An example of using **CONVERT_R** is for street aliases. For example, the file ST_ALIAS.TBL contains the following entries:

```
OBT "ORANGE BLOSSOM TRL"
SMF "SANTA MONICA FREEWAY"
WBN "WILSHIRE BLVD NORTH"
WBS "WILSHIRE BLVD SOUTH"
```

The pattern action set looks like:

```
*+
CONVERT_R [1] @ST_ALIAS.TBL
```

The alias is expanded and its individual tokens properly classified. Using the preceding example, WBN is expanded into three tokens with classes ?, T, and D. The remaining pattern and actions sets work as intended on the new address string.

Retyping tokens:

An optional token class can follow the TKN keyword.

The token is retyped to this class if the conversion is successful. If no class is specified, the token retains its original class. The following example converts a single unknown alphabetic token based on the entries in the three files. If there is a successful conversion, the token is retyped to either C, T, or U.

```
+
CONVERT [1] @CITIES.TBL TKN C
CONVERT [1] @TOWNS.TBL TKN T
CONVERT [1] @UNINCORP.TBL TKN U
```

Conversion actions for languages that are not Latin-based

You can use conversion with records that contain characters from languages that are not Latin-based. These actions can be used in data quality stages such as the Investigate stage and Standardize stage.

These conversion actions can be applied to any type of character. However, the actions are most useful when they are applied to languages that are processed by the CJK tokenizer, such as languages that are spoken in China, Japan, and Korea.

Converting Katakana and Hiragana characters:

You can convert Katakana and Hiragana characters by using the **TRANS_KH** and **TRANS_HK** actions.

The **TRANS_KH** action converts Katakana characters to Hiragana characters. The **TRANS_HK** action converts Hiragana characters to Katakana characters.

If you use these actions, specify the CJK tokenizer by using the **TOK** command in the PRAGMA section of the pattern-action file. The CKJ tokenizer handles tokens based on the conventions of languages that are not Latin-based.

Although the syntax for these actions is similar to the syntax for the **CONVERT** action, these actions do not use lookup tables and cannot convert an object to a literal. For example, you can use the following syntax for the **TRANS_KH** action:

```
TRANS_KH source TKN | TEMP retype1
```

Argument	Description
source	The object that is converted. The source can be an operand, dictionary field, or user variable.
retype1	The token class that you want assigned to the converted token. This argument is optional.

Converting character width:

You can convert the width of characters by using the **TRANS_WN** and **TRANS_NW** actions.

The **TRANS_WN** action converts full-width characters to half-width characters. The **TRANS_NW** action converts half-width characters to full-width characters.

If you use these actions, specify the CJK tokenizer by using the **TOK** command with an appropriate locale variable in the PRAGMA section of the pattern-action file. The CKJ tokenizer handles tokens based on the conventions of languages that are not Latin-based. For example, to use the **TRANS_WN** action to convert full-width Japanese characters to half-width characters, specify `TOK ja_JP`.

Although the syntax for these actions is similar to the syntax for the **CONVERT** action, these actions do not use lookup tables and cannot convert an object to a literal. For example, you can use the following syntax for the **TRANS_WN** action:

```
TRANS_WN source TKN | TEMP retype1
```

Argument	Description
source	The object that is converted. The source can be an operand, dictionary field, or user variable.
retype1	The token class that you want assigned to the converted token. This argument is optional.

CONVERT considerations

A CONVERT source can be an operand, a dictionary column, or a user variable, and if the source is an operand, it requires a third argument. The results of CONVERT_P, CONVERT_PL, and CONVERT_S actions can vary based on the pattern classes the actions are used with.

Some considerations to take into account when using the **CONVERT** action include:

- The source of a **CONVERT** can be an operand, a dictionary field, or a user variable. In the following example, both actions are valid:

  ```
  CONVERT temp @CODES.TBL
  CONVERT {CT} @CODES.TBL
  ```

- Entire path names can be coded for the convert table file specification:

  ```
  CONVERT {CT} @..\cnvrt\cnvrtfile.dat
  ```

- If the source of a **CONVERT** is an operand, a third argument is required:

  ```
  CONVERT operand table TKN
  CONVERT operand table TEMP
  ```

TKN is used to make the change permanent for all pattern action sets that follow the conversion and that involve this record.

If TEMP is included, the conversion applies only to the current set of actions.

If the conversion must be applied both to actions further down the program and to the current set of actions, specify two **CONVERT** actions (one by using TKN for other action sets and rule sets, and one by using TEMP for other actions in the current action set).

The results of CONVERT_P, CONVERT_PL, and CONVERT_S actions are affected by pattern classes. For example, the ? class can match to more than one token. If SUFFIX.TBL has the following lines:

```
STRASSE STRASSE 800.
AVENUE AVE 800.
```

If the pattern and actions are:

```
^ | ?
COPY [1] {HouseNumber}
CONVERT_S [2] @SUFFIX.TBL TEMP
COPY_S [2] {StreetName}
EXIT
```

The following input:

```
123 AAVENUE BB CSTRASSE
```

has the following result in the house number and street name fields:

```
123 AAVENUEBBC STRASSE
```

If the pattern and actions are:

```
^ | ?
CONVERT_S [2] @SUFFIX.TBL TKN T

^ | + | T | + | + | T
COPY [1] {HouseNumber}
COPY [5] {StreetName}
COPY [6] {StreetSuffixType}
EXIT
```

The following input:

```
123 AAVENUE BB CSTRASSE
```

results in:

123	C	STRASSE

The {HouseNumber}, {StreetName}, and {StreetSuffixType} fields are:

```
COPY [1] {HouseNumber}
COPY [2] {StreetName}
COPY [3] {StreetSuffixType}
```

which results in moving:

123	A	AVENUE

When you concatenate alphabetic characters with the pattern ?, the CONVERT_P, CONVERT_PL, or CONVERT_S action operates on the entire concatenated string for user variables, dictionary fields, and operands if the mode is TEMP.

For operands with a mode of TKN, each token in the token table that comprises the operand is examined individually and new tokens corresponding to the prefix or suffix are inserted into the table each time the prefix or suffix in question is found.

Retyping operands

The **RETYPE** action is used to change the class, token value, and abbreviation of an operand.

The format of the retype operand is the following pattern-action set:

```
RETYPE operand class [variable | literal] [variable | literal]
```

Argument	Description
operand	The operand number in the pattern
class	The new class value
variable | literal	The new token value, which can be a variable or a literal (optional)
variable | literal	The new token abbreviation can be a variable or a literal (optional) **Note:** If you want to change the token abbreviation but leave the token value as is, you can copy the token value to a user variable and use that variable as the third argument.

A basic concept of writing standardization rules is filtering. You can change phrases and clauses by detecting, processing, or removing them from the token table. You can remove them by retyping them to the NULL class (0). NULL classes are ignored in all pattern matching.

For example, if you want to process apartments and remove the unit information from further processing, you can use the following pattern action set:

```
*U | &
COPY_A [1] {UnitType}
COPY [2] {UnitValue}
RETYPE [1] 0
RETYPE [2] 0
```

Removing the apartment designation converts the address 123 MAIN ST APT 56 to a standard form, such as 123 MAIN ST. An apartment followed by any single token is detected. The fields are moved to the dictionary field and retyped to NULL, so that they would not match in any later patterns.

You can use a third operand to replace the text of a token. For example, if you want to recognize streets names like ST CHARLES and replace the ST with the word SAINT, you can use the following rule:

```
*!? | T | + | T
RETYPE [2] ? "SAINT"
```

This set scans for a type T token (the only value for type T is ST) preceded by any token type except unknown alphabetic character and followed by a single alphabetic word. The **RETYPE** action changes the type T operand to an unknown alphabetic character (?) and replaces the text with SAINT. If the input data is the following address:

```
123 ST CHARLES ST
```

The result corrects the ambiguity of the abbreviations for SAINT and STREET as shown in the following example:

```
123 SAINT CHARLES ST
```

This rule matches the standard ^ | ? | T pattern after the final ST is retyped to T as is done in the geocode rule set. The ? is used for the retype class. This is important because you want to consider SAINT to be the same token as the neighboring unknown alphabetic tokens.

The first operand of the pattern (*!?) makes sure that ST is not preceded by an unknown alphabetic character. This prevents the input MAIN ST JUNK from being standardized into MAIN SAINT JUNK.

A fourth operand is available for the **RETYPE** action to change the standard abbreviation of the operand. If you include this operand, you must also include the token value replacement argument (third argument). If you do not want to replace the token value, you can use the original value as the replacement value. For example, the address ST 123 can mean SUITE 123.

If the standard abbreviation for SUITE is STE, you need to change the token abbreviation with the following pattern-action set:

```
S | ^
RETYPE [1] U "SUITE" "STE"
```

This is interpreted in future patterns as SUITE 123 and has the abbreviation STE.

For the optional third and fourth arguments, you can use a substring range. For example, with an input record of 8 143rd Ave and the following pattern-action set:

```
^ | > | T
COPY [2](n) temp
RETYPE [2] ^ temp(1:2)
```

The 143 is copied to the variable *temp*, the 14 replaces the contents of the second token, and its class becomes numeric (^).

RETYPE reclassifies all elements of a possibly concatenated alphabetic class (?) or universal class (**), if no subfield ranges (n:m) are specified. **RETYPE** reclassifies only those tokens within the subfield range if a range is specified. For example, if you have the following input:

```
15 AA BB CC DD EE FF RD
```

The following patterns or actions have the described effect:

Pattern	Action	Effect
^ | ? | T	RETYPE [2] 0	; Sets AA to FF to NULL class
^ | 3 | T	RETYPE [2] 0	; Sets CC to NULL class

Pattern	Action	Effect
^ | (2:3) | T	RETYPE [2] 0	; Sets BB and CC to NULL class
^ | -2 | T	RETYPE [2] 0	; Sets EE to NULL class

RETYPE operates in a similar fashion to **CONVERT** with a TKN argument. The values for **RETYPE** (class, token value, abbreviation) are not available within the current pattern-action set and are available only for following pattern-action sets. Consequently, a pattern action set does not produce the wanted results and is therefore flagged as an error, as in the following example:

```
...
RETYPE [1] ...
COPY [1] ...
```

Retyping multiple tokens

A special pattern-action set is available for retyping multiple occurrences of a token.

The pattern must have the following format followed by a standard **RETYPE** statement referencing operand [1]:

```
number*class
```

The *number* must be an integer from 0 to MAX_TOKENS (currently 40) and indicates the number of referenced occurrences. Zero means that all occurrences are scanned.

For example, if you processed hyphenated house numbers, like 123-45 Main St., and you want to remove all hyphens from the tokens that remain, use the following pattern action set:

```
0 * -
RETYPE [1] 0
```

This scans for all hyphens and retypes them to NULL. If you want to retype two numeric tokens to an unknown type with a value of UKN, you use the following pattern-action set:

```
2 * ^
RETYPE [1] ? "UKN"
```

The entire pattern is restricted to this simple format when multiple occurrences are referenced.

Patterning

The PATTERN statement generates the pattern for the active token set.

The following pattern action shows the syntax:

```
PATTERN Target
```

Argument	Description
Target	The name of either a user variable or dictionary column where the returned patterns are stored

For example:

```
&
PATTERN {InputPattern}
```

In the previous example, when the **PATTERN** action executes, the pattern associated with all active tokens is generated and stored in the dictionary field {InputPattern} .

User overrides for domain preprocessor rule sets

You can create overrides for domain preprocessor (PREP) rule sets by using user override options.

Return code values

The overrides statements must be in your pattern action table. The statements enable the overrides in the Designer client.

Recommended syntax:

```
OVERRIDE_P Lookup_Value @Table_Name Return_Code [ CLASS | VALUE ]
```

Argument	Description
Lookup_Value	The name of the user variable containing either a pattern or text reference to the active token set that is to be checked for any applicable user overrides by looking up to the *Table_Name* provided
Table_Name	The name of the user override table to use
Return_Code	The name of either a user variable or dictionary field in which to store the return code value
[*CLASS* | *VALUE*]	*CLASS* indicates that the *Lookup_Value* contains a pattern reference where each token is represented by its token class. *VALUE* indicates that the *Lookup_Value* contains a text reference where each token is represented by its token value.

Value	Description
0 (zero) (default value)	No user overrides were executed because no match was found for the specified *Lookup_Value* on the specified *Table_Name*
1	A user override was successfully executed with no errors

For example:

```
OVERRIDE_P Text @USPREPIP.TBL Return VALUE
OVERRIDE_P Pattern @USPREPIP.TBL Return CLASS
```

The specified *Lookup_Value* is searched for on the specified *Table_Name*. If the *Lookup_Value* is found, the active token set is **RETYPE** that uses the override instructions found in the matching table entry. A return code value is always returned and stored in the specified *Return_Code*.

Lookup_Value can contain either a pattern or a string of literal token values, which represents the entire active token set.

Table_Name is a two-column STAN conversion table that uses the following formats:

- Pattern Override Table Format (for domain preprocessor rule sets)

  Column 1: <[Classification for Token #1][Classification for Token #2]...>

  Column 2: <[Domain Mask for Token #1][Domain Mask for Token #2]...>

  For example (for domain preprocessor rule sets):

```
N^D+TU^   AAAAAAA
A++E      NNNN
A+S^      RRRR
```

- Text Override Table Format (for domain preprocessor rule sets)

  Column 1: <["][Token #1][space][Token #2][space]...["]>

  Column 2: <[Domain Mask for Token #1][Domain Mask for Token #2]...>

  For example (for domain preprocessor rule sets):

```
"ZQNAMEZQ 456 SOUTH MAIN AVENUE APARTMENT 7"   AAAAAA
"ZQADDRZQ IBM CORPORATION"   NNNN
"ZQADDRZQ LITTLETON MA 01460"   RRRR
```

For more information on domain preprocessor rule sets, see the *IBM InfoSphere QualityStage User's Guide*.

Override table syntax validation

The syntax validation of the override tables occurs during Standardize or Investigation stage initialization.

If any syntax errors are found, the stage execution stops and an error message is written to the log file including the override table name, the contents of the invalid entries, and an explanation of the nature of the errors.

For each active token there should be one corresponding domain mask in the override instruction. The domain mask is the class to use to **RETYPE** the corresponding token. The only valid values for a domain mask are A, N, and R.

The two primary errors to check for are:

- Invalid domain masks (not A, N, or R)
- Incomplete override instructions (there was not a valid instruction for each active token)

User overrides for rule sets

You can create overrides by using user override options within domain rule sets.

Return code values

The overrides statements must be in your pattern action table. The statements enable the overrides in the Designer client.

You can use the following syntax:

```
OVERRIDE_D Lookup_Value @Table_Name Return_Code [ CLASS | VALUE ]
```

Argument	Description
Lookup_Value	The name of the user variable containing either a pattern or text reference to the active token set that is to be checked for any applicable user overrides by looking up to the *Table_Name* provided.
Table_Name	The name of the user override table to use.
Return_Code	The name of either a user variable or dictionary field in which to store the return code value.
[CLASS | VALUE]	CLASS indicates that the *Lookup_Value* contains a pattern reference where each token is represented by its token class. VALUE indicates that the *Lookup_Value* contains a text reference where each token is represented by its token value.

Value	Description
0 (zero) (default value)	No user overrides were executed because no match was found for the specified *Lookup_Value* on the specified *Table_Name*.
1	A user override was successfully executed with no errors.

An example of the user created override action:

```
OVERRIDE_D Text @USNAME.UTO Return VALUE
OVERRIDE_D Pattern @USNAME.UPO Return CLASS
```

The specified *Lookup_Value* is searched for on the specified *Table_Name*. If the *Lookup_Value* is found, the active token set is written to dictionary fields by using the override instructions found in the matching table entry and then **RETYPE**d to null. Any existing content of the dictionary field is preserved while running the override instructions. A return code value is always returned and stored in the specified *Return_Code*.

The *Lookup_Value* can contain either a pattern or a string of literal token values, which represents the entire active token set.

The *Table_Name* is a two-column STAN conversion table that uses the following formats:

- Pattern Override Table Format (for rule sets)

  Column 1: <[Classification for Token #1][Classification for Token #2]...>

  Column 2: <[DCT Field for Token #1][Data Flag for Token #1] - [DCT Field for Token #2][Data Flag for Token #2] -...>

  For example (for rule sets):

  ```
  FIF   FirstName1-MiddleName1-PrimaryName1
  ^D+T   HouseNumber1
  -StreetPrefixDirectional2-StreetName1-StreetSuffixType2
  +++   ExceptionData5
  ```

- Text Override Table Format (for rule sets)

  Column 1: <["][Token #1][space][Token #2][space]...["]>

Column 2: <[DCT Field for Token #1][Data Flag for Token #1] [DCT Field for Token #2] [Data Flag for Token #2]...>

Text Overrides Examples (for rule sets)

```
"JAMES E HARRIS" FirstName1-MiddleName1-PrimaryName1
"123 N. MAIN STREET" HouseNumber1
-StreetPrefixDirectional2-StreetName1-StreetSuffixType2
"DO NOT MAIL"   ExceptionData5
```

For more information on categories of rule sets, including domain-specific see the *IBM InfoSphere QualityStage User's Guide*.

Override table syntax validation for domain-specific rule sets

The syntax validation of the override tables is performed during Standardize or Investigation stage initialization.

If any syntax errors are found, the stage execution stops and an error message is written to the log file including the override table name, the contents of the invalid entries, and an explanation of the nature of the errors.

For each active token there is a corresponding instruction in the override statement. The instruction consists of the field named followed by a data flag. A valid data flag indicates the associated actions to be performed on the corresponding token. Each override instruction is separated by a hyphen (-). The only situation where there are fewer valid override instructions than active tokens is when a "move all remaining tokens" or "drop all remaining tokens" data flag is used (values 5,6,7,8, and 9 in the following table.

The three primary errors to check for are:

- Invalid data flags (not a value listed in the following the table)
- Invalid dictionary field names (not listed in the dictionary file of the rule set)
- Incomplete override instructions (there was not a valid instruction for each active token)

Data flag table (used for domain-specific rule sets only):

Value	Associated actions
0 (zero)	Drop the current token
1	Append a leading character space, then append the original value of the corresponding token, to the specified dictionary field (no leading character space can be appended if the specified dictionary field is empty)
2	Append a leading character space, then append the standard value of the corresponding token, to the specified dictionary field (no leading character space can be appended if the specified dictionary field is empty, also if no standard value is available for the corresponding token then use the original value)
3	Append the original value of the corresponding token, without appending a leading character space, to the specified dictionary field

Value	Associated actions
4	Append the standard value of the corresponding token, without appending a leading character space, to the specified dictionary field (if no standard value is available for the corresponding token then use the original value)
5	Move all remaining tokens that use their original values, leaving one character space between each token, to the specified dictionary field
6	Move all remaining tokens that use their standard values, leaving one character space between each token, to the specified dictionary field (if no standard value is available for any of the remaining tokens then use the original value)
7	Move all remaining tokens that use their original values, with no character space between each token, to the specified dictionary field
8	Move all remaining tokens that use their standard values, with no character space between each token, to the specified dictionary field (if no standard value is available for any of the remaining tokens then use the original value)
9	Drop all remaining tokens

Setting margins

You can control what portion of the pattern is available to match to by setting left and right margins.

You might want to restrict pattern-action sets to ranges of input tokens. For example, if your input consists of records each consisted of four lines, two lines of address information and zero, one, or two lines of department or institution names, you might want to search for an address line with a number, street name, and street type, process only that line, and retype the tokens to NULL.

If you did the same for the city, state, and postal line, you can be guaranteed that anything remaining is department or institution names.

This type of restricted processing can be accomplished by setting left and right margins. The default settings are: left margin is the first token and right margin is the last token. These settings can be changed by using the **SET_L_MARGIN** and **SET_R_MARGIN** actions. Each action takes one of two keywords followed by a number or number within square brackets. You can use the following syntax:

```
SET_L_MARGIN LINE number | OPERAND number
SET_R_MARGIN LINE number | OPERAND number
```

An example of restricting the action sets is as follows:

```
*^ | D | ?
SET_L_MARGIN OPERAND [3]
SET_R_MARGIN OPERAND [3]
```

In this example, both the left and right margins are set to operand three. Since more than one token can comprise the token set corresponding to the unknown alphabetic type ?, the left margin is always set to the first token of the set and the right margin is always set to the last token of the set.

The same holds true for the type **. If operand three has specified another type that can only refer to a single token, such as ^, setting both left and right margins to operand three results in the next pattern-action sets only seeing that single token. As with any pattern-action set, if the associated pattern does not match to the input record, the actions are ignored and the margins are left unchanged.

After processing is complete within the set of tokens specified by the left and right margins, you need to reset the margins to their widest setting to reposition the other lines or tokens. Use the keywords **BEGIN_TOKEN** for the left margin and **END_TOKEN** for the right margin:

```
SET_L_MARGIN OPERAND [BEGIN_TOKEN]
SET_R_MARGIN OPERAND [END_TOKEN]
```

It is important to note that if, after setting margins, you retype all tokens within the margins to NULL, you have no access to any other tokens. To execute the actions that reset the margins, you must use a pattern that does not require any token to be found. The normal method is, at the beginning of a process, to create a condition that tests true at any point in the pattern-action sequence. Your first pattern-action set could be:

```
&
COPY "TRUE" true
```

At some point after all visible tokens have been retyped to NULL, you can execute the margin resetting actions by using the following pattern-action sequence:

```
[true = "TRUE"]
SET_L_MARGIN OPERAND [BEGIN_TOKEN]
SET_R_MARGIN OPERAND [END_TOKEN]
```

SOUNDEX phonetic coding

The **SOUNDEX** action is used to compute a **SOUNDEX** code of a dictionary field and move the results to another dictionary field.

SOUNDEX codes are phonetic keys that are useful for blocking records in a matching operation.

SOUNDEX is an excellent blocking variable because it is not very discriminating and yet is used to partition the files into a reasonable number of subsets. The action is specifically designed for the English language but perhaps can be useful for languages.

The **SOUNDEX** action has the following format:

```
SOUNDEX source-field target-field
```

For example, the following action computes the **SOUNDEX** of the StreetName dictionary field and places the result in the field: **StreetNameSOUNDEX** field:

```
SOUNDEX {StreetName} {StreetNameSOUNDEX}
```

The **RSOUNDEX** (reverse **SOUNDEX**) action is the same as the **SOUNDEX** action except that the phonetic code is generated from the last non-blank character of the field and

proceeds to the first. This is useful for blocking fields where the beginning characters are in error. An example of the pattern syntax is as follows:

```
RSOUNDEX {StreetName} {StreetNameRVSNDX}
```

The **SOUNDEX** and **RSOUNDEX** actions are used in the **POST** action section, so they are executed after pattern matching is complete for the record.

POST actions must occur before any pattern-action sets and are preceded by the line **POST_START** and followed by the line **POST_END**.

NYSIIS coding

The NYSIIS action is used to compute the NYSIIS code of a dictionary field and move the results to another dictionary field.

NYSIIS stand for New York State Information and Intelligence Systems. NYSIIS is an phonetic coding algorithm developed after SOUNDEX. Similar to SOUNDEX, NYSIIS is specifically designed for the English language but perhaps can be useful for languages.

The NYSIIS action has the following format:

```
NYSIIS source-field target-field
```

For example, the following action computes the NYSIIS of the MatchFirstName field and places the eight character result in the MatchFirstNameNYSIIS dictionary field:

```
NYSIIS {MatchFirstName} {MatchFirstNameNYSIIS}
```

The RNYSIIS action is reverse NYSIIS. The NYSIIS and RNYSIIS actions are used in the **POST** action section, so they are run after pattern matching is complete for the record.

POST actions must occur before any pattern-action sets and must be preceded by the line **POST_START** and followed by the line **POST_END**.

Terminating pattern matching

The **EXIT** action is used to quit the pattern matching program for the current process.

The **EXIT** action prevents further pattern-action sets from being run; for example:

```
^ | D | ? | T
COPY [1] {HouseNumber}
COPY_A [2] {StreetPrefixDirectional}
COPY [3] {StreetName}
COPY [4] {StreetSuffixType}
EXIT
```

If the input record matches the pattern, the current process ends after the **COPY** actions (and after any **POST** actions).

Calling subroutines

Subroutines can be used to improve processing time and control the number of pattern action sets that need to be coded.

Subroutines are invoked with the syntax:

```
CALL <subroutine name>
```

Because pattern action sets are executed sequentially, it is fastest to test for a generic type and call a subroutine to process that type. This is best illustrated by an example:

```
*U
CALL UNITS
```

If a unit type (U) is detected anywhere in the input, the subroutine Units is called to process the apartment numbers. Subroutine names are formed according to the same rules as variables names (up to 32 characters with the first character being alphabetic).

If a subroutine that does not exist is called, the rule set aborts at runtime.

Returning from a subroutine

The **RETURN** action is available to return control from a subroutine to the main program. A **RETURN** is not required immediately preceding the \END_SUB statement.

Writing subroutines

A subroutine is delimited like the **POST** actions.

Subroutines have a header and a trailer line:

```
\SUB subroutine_name
subroutine_body
\END_SUB
```

You can add as many **SUB** and **END_SUB** sets as required. The following example illustrates a rules file organization:

```
\POST_START
actions
\POST_END
%1 U
CALL UNITS
%1 R
CALL ROUTES
additional_patterns_and_actions
\SUB UNITS
apartment_processing_patterns_and_actions
\END_SUB
\SUB ROUTES
route_processing_patterns_and_actions
\END_SUB
```

All subroutines are coded at the end of the file. The order of the subroutines themselves is unimportant. The subroutines contain the standard pattern action sets as found in the main rules.

Subroutines can be nested. That is, **CALL** actions are permitted within subroutines.

Control is returned back to the level from which a routine was called, either another subroutine or the main program, when the **END_SUB** is reached or when a **RETURN** action is executed.

Performing actions repetitively

Use the **REPEAT** action to execute the preceding pattern and its associated action set repetitively until the pattern fails to match.

When you use the **REPEAT** action, the **REPEAT** statement must be the last action statement in the action set. The **REPEAT** statement is valid after a **CALL** statement. The action set must include one or more actions that change the objects of the pattern test or an indefinite loop results.

In the following example, the pattern action set changes the types of all operands with class A to the unknown alpha class (?) after converting the input value to the standard value::

```
*A
COPY_A [1] temp
RETYPE [1] ? temp temp
REPEAT
```

In this example, the pattern action set calls the Parsing_Rules subroutine until the user variable `data` is empty:

```
[data != ""]
CALL Parsing_Rules
REPEAT
```

You can see an example of the Parsing_Rules subroutine in the USNAME rule set. In the Designer client repository view, expand the **Standardization Rules** folder. Expand USA domain folder.

Summary of sources and targets

The following is a summary of the sources and targets allowed for all actions.

Action	Source	Target
CONCAT	operand operand literal literal user variable user variable	user variable dictionary field user variable dictionary field dictionary field dictionary variable
CONCAT_A	operand operand	user variable dictionary field
CONCAT_I	operand operand literal literal user variable user variable	user variable dictionary field user variable dictionary field dictionary field user variable
CONVERT	dictionary field user variable operand	The CONVERT statements convert the source. The source is also the target.

Action	Source	Target
CONVERT_P | PL | S	dictionary field user variable operand	The CONVERT statements convert the source. The source is also the target.
CONVERT_R	operand	The CONVERT statements convert the source. The source is also the target.
COPY	operand operand formatted field dictionary field dictionary field literal literal user variable user variable	dictionary field user variable dictionary field dictionary field user variable dictionary field user variable dictionary field user variable
COPY_A	operand operand	user variable dictionary field
COPY_C	operand operand	dictionary field user variable
COPY_I	operand operand formatted field dictionary field dictionary field literal literal user variable user variable	dictionary field user variable dictionary field dictionary field user variable dictionary field user variable dictionary field user variable
COPY_S	operand operand	dictionary field user variable
MOVE	user variable dictionary field	dictionary field dictionary field
NYSIIS	dictionary field user variable	dictionary field dictionary field
PREFIX	operand operand literal literal user variable user variable	user variable dictionary field user variable dictionary field dictionary field user variable
PREFIX_A	operand operand	user variable dictionary field

Action	Source	Target
PREFIX_I	operand operand literal literal user variable user variable	user variable dictionary field user variable dictionary field dictionary field user variable
REPEAT	(none)	(none)
RNYSIIS	dictionary field user variable	dictionary field dictionary field
RSOUNDEX	dictionary field user variable	dictionary field dictionary field
SOUNDEX	dictionary field user variable	dictionary field dictionary field
TRANS_HK|KH	dictionary field user variable operand	These statements convert the source. The source is also the target.
TRANS_NW|WN	dictionary field user variable operand	These statements convert the source. The source is also the target.

Margin setting actions have no source or target, instead they take a first argument of keyword OPERAND/LINE and a second argument of operand number (enclosed in square brackets []) or line number, for example:

```
SET_L_MARGIN LINE 5
SET_R_MARGIN OPERAND [3]
```

Accessing product documentation

Documentation is provided in a variety of locations and formats, including in help that is opened directly from the product client interfaces, in a suite-wide information center, and in PDF file books.

The information center is installed as a common service with IBM InfoSphere Information Server. The information center contains help for most of the product interfaces, as well as complete documentation for all the product modules in the suite. You can open the information center from the installed product or from a Web browser.

Accessing the information center

You can use the following methods to open the installed information center.

- Click the **Help** link in the upper right of the client interface.

  Note: From IBM InfoSphere FastTrack and IBM InfoSphere Information Server Manager, the main Help item opens a local help system. Choose **Help > Open Info Center** to open the full suite information center.
- Press the F1 key. The F1 key typically opens the topic that describes the current context of the client interface.

  Note: The F1 key does not work in Web clients.
- Use a Web browser to access the installed information center even when you are not logged in to the product. Enter the following address in a Web browser: http://host_name:port_number/infocenter/topic/com.ibm.swg.im.iis.productization.iisinfsv.home.doc/ic-homepage.html. The host_name is the name of the services tier computer where the information center is installed, and port_number is the port number for InfoSphere Information Server. The default port number is 9080. For example, on a Microsoft® Windows® Server computer named iisdocs2, the Web address is in the following format: http://iisdocs2:9080/infocenter/topic/com.ibm.swg.im.iis.productization.iisinfsv.nav.doc/dochome/iisinfsrv_home.html.

A subset of the information center is also available on the IBM Web site and periodically refreshed at http://publib.boulder.ibm.com/infocenter/iisinfsv/v8r7/index.jsp.

Obtaining PDF and hardcopy documentation

- A subset of the PDF file books are available through the InfoSphere Information Server software installer and the distribution media. The other PDF file books are available online and can be accessed from this support document: https://www.ibm.com/support/docview.wss?uid=swg27008803&wv=1.
- You can also order IBM publications in hardcopy format online or through your local IBM representative. To order publications online, go to the IBM Publications Center at http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss.

Providing feedback about the documentation

You can send your comments about documentation in the following ways:

- Online reader comment form: www.ibm.com/software/data/rcf/
- E-mail: comments@us.ibm.com

Contacting IBM

You can contact IBM for customer support, software services, product information, and general information. You also can provide feedback to IBM about products and documentation.

The following table lists resources for customer support, software services, training, and product and solutions information.

Table 3. IBM resources

Resource	Description and location
IBM Support Portal	You can customize support information by choosing the products and the topics that interest you at www.ibm.com/support/entry/portal/Software/Information_Management/InfoSphere_Information_Server
Software services	You can find information about software, IT, and business consulting services, on the solutions site at www.ibm.com/businesssolutions/
My IBM	You can manage links to IBM Web sites and information that meet your specific technical support needs by creating an account on the My IBM site at www.ibm.com/account/
Training and certification	You can learn about technical training and education services designed for individuals, companies, and public organizations to acquire, maintain, and optimize their IT skills at http://www.ibm.com/software/sw-training/
IBM representatives	You can contact an IBM representative to learn about solutions at www.ibm.com/connect/ibm/us/en/

Providing feedback

The following table describes how to provide feedback to IBM about products and product documentation.

Table 4. Providing feedback to IBM

Type of feedback	Action
Product feedback	You can provide general product feedback through the Consumability Survey at www.ibm.com/software/data/info/consumability-survey

Table 4. Providing feedback to IBM (continued)

Type of feedback	Action
Documentation feedback	To comment on the information center, click the Feedback link on the top right side of any topic in the information center. You can also send comments about PDF file books, the information center, or any other documentation in the following ways: • Online reader comment form: www.ibm.com/software/data/rcf/ • E-mail: comments@us.ibm.com

Product accessibility

You can get information about the accessibility status of IBM products.

The IBM InfoSphere Information Server product modules and user interfaces are not fully accessible. The installation program installs the following product modules and components:

- IBM InfoSphere Business Glossary
- IBM InfoSphere Business Glossary Anywhere
- IBM InfoSphere DataStage
- IBM InfoSphere FastTrack
- IBM InfoSphere Information Analyzer
- IBM InfoSphere Information Services Director
- IBM InfoSphere Metadata Workbench
- IBM InfoSphere QualityStage

For information about the accessibility status of IBM products, see the IBM product accessibility information at http://www.ibm.com/able/product_accessibility/index.html.

Accessible documentation

Accessible documentation for InfoSphere Information Server products is provided in an information center. The information center presents the documentation in XHTML 1.0 format, which is viewable in most Web browsers. XHTML allows you to set display preferences in your browser. It also allows you to use screen readers and other assistive technologies to access the documentation.

IBM and accessibility

See the IBM Human Ability and Accessibility Center for more information about the commitment that IBM has to accessibility.

Notices and trademarks

This information was developed for products and services offered in the U.S.A.

Notices

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies:

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office

UNIX is a registered trademark of The Open Group in the United States and other countries.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The United States Postal Service owns the following trademarks: CASS, CASS Certified, DPV, LACS^{Link}, ZIP, ZIP + 4, ZIP Code, Post Office, Postal Service, USPS and United States Postal Service. IBM Corporation is a non-exclusive DPV and LACS^{Link} licensee of the United States Postal Service.

Other company, product or service names may be trademarks or service marks of others.

Index

Q

R

S

T

U

Printed in USA

SC19-3480-00