MATCH

Match provides a fairly general pattern match facility that allows you to specify certain tests that would otherwise be clumsy to write, by giving a pattern which the datum is supposed to match.

Essentially, you write "Does the (expression) X look like (the pattern) P?"

For example, `(MATCH X WITH (& 'A -- 'B))` asks whether the second element of `X` is an `A`, and the last element a `B`.

Requirements

DWIM must be enabled.

Installation

Load `MATCH.LCOM` from the library.

Programmer's Interface

`(MATCH` *OBJECT* `WITH` *PATTERN*`)` [CLISP operator]

Matches the *OBJECT* with the *PATTERN*.

The implementation of the matching is performed by computing (once) the equivalent Lisp expression whichperforms the indicated operation, and substituting this for the pattern (rather than by invoking each time a general purpose capability such as that found in the AI languages FLIP or PLANNER).

For example, the translation of

```
(MATCH X WITH (& 'A -- 'B)) is:

(AND (EQ (CADR X) 'A)
     (EQ (CAR (LAST (CDDR X))) 'B))
```

Thus the pattern match facility is really a pattern match compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g., in inner loops. Wherever possible, already existing Lisp functions are used in the translation, e.g., the translation of `($ 'A $)` uses `MEMB`, `($ ('A $) $)` uses `ASSOC`, etc.

The syntax for pattern match expressions is `(MATCH` *FORM* `WITH` *PATTERN*`)`, where *PATTERN* is a list as described below. If *FORM* appears more than once in the translation, and it is not either a variable or an expression that is easy to (re)compute, such as `(CAR Y)`, `(CDDR Z)`, etc., a dummy variable is generated and bound to the value of *FORM* so that *FORM* is not evaluated a multiple number of times.

For example, the translation of

```
(MATCH (FOO X) WITH ($ 'A $))
```

is simply

```
(MEMB 'A (FOO X)),
```

while the translation of

```
(MATCH (FOO X) WITH ('A 'B --)) is:

[PROG ($$2)
    (RETURN
        (AND (EQ (CAR (SETQ $$2 (FOO X))) 'A)
             (EQ (CADR $$2) 'B]
```

In the interests of efficiency, the pattern match compiler assumes that all lists end in `NIL`, i.e., there are no LISTP checks inserted in the translation to check tails.

For example, the translation of

```
(MATCH X WITH ('A & --))
```

is

```
(AND (EQ (CAR X) (QUOTE A)) (CDR X)),
```

which matches with `(A B)` as well as `(A . B)`.

Similarly, the pattern match compiler does not insert LISTP checks on elements, e.g.,

```
(MATCH X WITH (('A --) --))
```

translates as

```
(EQ (CAAR X) 'A),
```

and

```
(MATCH X WITH (($1 $1 --) --))
```

translates as

```
(CDAR X)
```

Note that you can explicitly insert LISTP checks yourself by using `@`, as described below, e.g.,

```
(MATCH X WITH (($1 $1 --)@LISTP --))
```

translates as

```
(CDR (LISTP (CAR X)))
```

`PATLISPCHECK` [Variable]

The insertion of LISTP checks for *ELEMENTS* is controlled by the variable `PATLISTPCHECK`. When `PATLISTPCHECK` is `T`, LISTP checks are inserted, e.g.,

```
(MATCH X WITH (('A --) --))
```

translates as:

```
(EQ (CAR (LISTP (CAR (LISTP X)))) 'A)
```

PATLISTPCHECK is initially NIL. Its value can be changed within a particular function by using a local CLISP declaration (see *IRM*).

PATVARDEFAULT [Variable]

Controls the treatment of !*ATOM* patterns (see below).

If PATVARDEFAULT is ' or QUOTE, !*ATOM* is treated the same as '*ATOM*.

If PATVARDEFAULT is = or EQUAL, same as =*ATOM*.

If PATVARDEFAULT is == or EQ, same as ==*ATOM*.

If PATVARDEFAULT is _ or SETQ, same as *ATOM*_&.

PATVARDEFAULT is initially ' (quote).

PATVARDEFAULT can be changed within a particular function by using a local CLISP declaration (see *IRM*).

Note: Numbers and strings are always interpreted as though PATVARDEFAULT were =, regardless of its setting. EQ, MEMB, and ASSOC are used for comparisons involving small integers.

Note: Pattern match expressions are translated using the DWIM and CLISP facilities, using all CLISP declarations in effect (standard/fast/undoable; see *IRM*).

Pattern Elements

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment.

For example, in the TTY editor's pattern matcher (see *IRM*), "--" matches any arbitrary segment of a list, while & or a subpattern match only one element of a list. Those patterns which may match a segment of a list are called segment patterns; those that match a single element are called element patterns.

Element Patterns

There are several types of element patterns, best given by their syntax:

$1 or &	Matches an arbitrary element of a list.
'*EXPRESSION*	Matches only an element which is equal to the given expression e.g., 'A, '(A B).
	EQ, MEMB, and ASSOC are automatically used in the translation when the quoted expression is atomic, otherwise EQUAL, MEMBER, and SASSOC.
=*FORM*	Matches only an element which is EQUAL to the value of *FORM*; e.g., =X, =(REVERSE Y).
==*FORM*	Same as =, but uses an EQ check instead of EQUAL.
ATOM	The treatment depends on setting of PATVARDEFAULT (see above).
(*PATTERN1* ... *PATTERNn*)	Matches a list which matches the given patterns; e.g.,

`(& &), (-- 'A).`

ELEMENT-PATTERN`@`*FN* Matches an element if *ELEMENT-PATTERN* matches it, and *FN* (name of a function or a LAMBDA expression) applied to that element returns non-`NIL`.

For example, `&@NUMBERP` matches a number, and `('A --)@FOO` matches a list whose first element is `A` and for which `FOO` applied to that list is non-`NIL`.

For simple tests, the function-object is applied before a match is attempted with the pattern, e.g.,

```
((-- 'A --)@LISTP --)
```

translates as

```
(AND  (LISTP (CAR X)) (MEMB 'A (CAR X))),
```

not the other way around. *FN* may also be a *FORM* in terms of the variable `@`, e.g., `&@(EQ @ 3)` is equivalent to `=3`.

`*` Matches any arbitrary element. If the entire match succeeds, the element which matched the * is returned as the value of the match.

Note: Normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-`NIL` if the match succeeds and `NIL` if it fails. However, if a * appears in the pattern, the expression generated could also return `NIL` if the match succeeds and * was matched to `NIL`.

For example,

```
(MATCH X WITH ('A * --))
```

translates as

```
(AND (EQ (CAR X) 'A) (CADR X)),
```

so if `X` is equal to `(A NIL B)` then `(MATCH X WITH ('A * --))` returns `NIL` even though the match succeeded.

~ELEMENT-PATTERN Matches an element if the element is not (~) matched by *ELEMENT-PATTERN*, e.g., `~'A, ~=X, ~(-- 'A --).`

`(*ANY*` *ELEMENT-PATTERN ELEMENT-PATTERN* `...)`

Matches if any of the contained patterns match.

Segment Patterns

`$` or `--` Matches any segment of a list (including one of zero length).

The difference between $ and -- is in the type of search they generate.

For example,

```
(MATCH X WITH ($ 'A 'B $))
```

translates as

```
(EQ (CADR (MEMB 'A X)) 'B)
```

whereas

```
(MATCH X WITH (-- 'A 'B $))
```

translates as:

```
[SOME X (FUNCTION (LAMBDA ($$2 $$1)
  (AND (EQ $$2 'A)
    (EQ (CADR $$1) 'B]
```

Thus, a paraphrase of `($ 'A 'B $)` would be "Is B the element following the first A?", whereas a paraphrase of `(-- 'A 'B $)` would be "Is there any A immediately followed by a B?"

Note that the pattern using `$` results in a more efficient search than that using `--`. However, `($ 'A 'B $)` does not match with `(X Y Z A M O A B C)`, but `(-- 'A 'B $)` does.

Essentially, once a pattern following a `$` matches, the `$` never resumes searching, whereas `--` produces a translation that always continues searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both `--` and `$` is the same.

For example, both

```
(MATCH X WITH ($ 'A $3 $))
```

and

```
(MATCH X WITH (-- 'A $3 --))
```

translate as

```
(CDDDR (MEMB (QUOTE A) X))
```

because if there are not three elements following the first `A`, there certainly will not be three elements following subsequent `A`'s, so there is no reason to continue searching, even for `--`.

Similarly, `($ 'A $ 'B $)` and `(-- 'A -- 'B --)` are equivalent.

`$2`, `$3`, etc. Matches a segment of the given length.

Note that `$1` is not a segment pattern.

`!`*ELEMENT-PATTERN* Matches any segment which *ELEMENT-PATTERN* would match as a list.

For example, if the value of `FOO` is `(A B C)`, `!=FOO` matches the segment ... `A B C` ... etc.

Note: Since `!` appearing in front of the last pattern specifies a match with some tail of the given expression, it also makes sense in this case for a `!` to appear in front of a pattern that can only match with an atom, e.g., `($2 !'A)` means match if `CDDR` of the expression is the atom A.

Similarly,

```
(MATCH X WITH ($ ! 'A))
```

translates to

```
(EQ (CDR (LAST X)) 'A).
```

!*ATOM* The treatment depends on setting of PATVARDEFAULT.

If PATVARDEFAULT is ' or QUOTE, same as !'*ATOM* (see above discussion).

If PATVARDEFAULT is = or EQUAL, same as !=*ATOM*.

If PATVARDEFAULT is == or EQ, same as !==*ATOM*.

If PATVARDEFAULT is _ or SETQ, same as *ATOM*_$.

. The atom "." is treated *exactly* like "!". In addition, if a pattern ends in an atom, the "." is first changed to "!", e.g., ($1 . A) and ($1 ! A) are equivalent, even though the atom "." does not explicitly appear in the pattern.

One exception where "." is not treated like "!" is when "." preceding an assignment does not have the special interpretation that "!" has preceding an assignment (see below).

For example,

```
(MATCH X WITH ('A . FOO_'B))
```

translates as:

```
(AND (EQ (CAR X) 'A)
     (EQ (CDR X) 'B)
     (SETQ FOO (CDR X)))
```

but

```
(MATCH X WITH ('A ! FOO_'B))
```

translates as:

```
(AND (EQ (CAR X) 'A)
     (NULL (CDDR X))
     (EQ (CADR X) 'B)
     (SETQ FOO (CDR X)))
```

SEGMENT-PATTERN@*FUNCTION-OBJECT*

Matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-NIL.

For example, ($@CDDR 'D $) matches (A B C D E) but not (A B D E), since CDDR of (A B) is NIL.

Note: An @ pattern applied to a segment requires computing the corresponding structure (with LDIFF) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

Assignments

Any pattern element may be preceded by "*VARIABLE*_", meaning that if the match succeeds (i.e., everything matches), *VARIABLE* is set to the thing that matches that pattern element.

For example, if X is `(A B C D E)`, `(MATCH X WITH ($2 Y_$3))` sets Y to `(C D E)`.

Note that assignments are not performed until the entire match has succeeded, so assignments cannot be used to specify a search for an element found earlier in the match. For example, `(MATCH X WITH (Y_$1 =Y --))` does not match with `(A A B C ...)`, unless, of course, the value of Y was A before the match started. This type of match is achieved by using place-markers, described below.

If the variable is preceded by a `!`, the assignment is to the tail of the list as of that point in the pattern, i.e., that portion of the list matched by the remainder of the pattern.

For example, if X is `(A B C D E)`, `(MATCH X WITH ($ !Y_'C 'D $))` sets Y to `(C D E)`, i.e., CDDR of X. In other words, when `!` precedes an assignment, it acts as a modifier to the _, and has no effect whatsoever on the pattern itself, e.g., `(MATCH X WITH ('A 'B))` and `(MATCH X WITH ('A !FOO_'B))` match identically, and in the latter case, FOO is set to CDR of X.

Note: `*`*_PATTERN-ELEMENT* and `!*`*_PATTERN-ELEMENT* are acceptable, e.g.,

```
(MATCH X WITH ($ 'A *_('B --) --))
```

translates as:

```
[PROG ($$2) (RETURN
(AND (EQ (CAADR (SETQ $$2 (MEMB 'A X)))  'B)
      (CADR $$2]
```

Place Markers

Variables of the form #*N*, where *N* is a number, are called place markers, and are interpreted specially by the pattern match compiler. Place markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e., they can be assigned values, or used freely in forms appearing in the pattern.

For example,

```
(MATCH X WITH (#1_$1 =(ADD1 #1)))
```

matches the list `(2 3)`.

However, they are not really variables in the sense that they are not bound, nor can a function called from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of PATVARDEFAULT, the first appearance of a defaulted place-marker is interpreted as though PATVARDEFAULT were _.

Thus the above pattern could have been written as

```
(MATCH X WITH ( 1 =(ADD1  1))).
```

Subsequent appearances of a place-marker are interpreted as though PATVARDEFAULT were =.

For example,

```
(MATCH X WITH (#1 #1 --))
```

is equivalent to

```
(MATCH X WITH (#1_$1 =#1 --))
```

and translates as

```
(AND (CDR X) (EQUAL (CAR X) (CADR X))
```

Note that `(EQUAL (CAR X) (CADR X))` would incorrectly match with `(NIL)`.

Replacements

The construct *PATTERN-ELEMENT_FORM* specifies that if the match succeeds, the part of the data that matched is to be replaced with the value of *FORM*.

For example, if `X =(A B C D E)`, `(MATCH X WITH ($ 'C $1_Y $1))` replaces the third element of `X` with the value of `Y`. As with assignments, replacements are not performed until after it is determined that the entire match is successful.

Replacements involving segments splice the corresponding structure into the list being matched, e.g., if `X` is `(A B C D E F)` and `FOO` is `(1 2 3)`, after the pattern `('A $_FOO 'D $)` is matched with `X`, `X` is `(A 1 2 3 D E F)`, and `FOO` is `EQ` to `CDR` of `X`, i.e., `(1 2 3 D E F)`.

Note that `($ FOO_FIE $)` is ambiguous, since it is not clear whether `FOO` or `FIE` is the pattern element, i.e., whether _ specifies assignment or replacement.

For example, if `PATVARDEFAULT` is =, this pattern can be interpreted as `($ FOO_=FIE $)`, meaning search for the value of `FIE`, and if found set `FOO` to it, or `($ =FOO_FIE $)` meaning search for the value of `FOO`, and if found, store the value of `FIE` into the corresponding position. In such cases, you should disambiguate by not using the `PATVARDEFAULT` option, i.e., by specifying `'` or =.

Note: Replacements are normally done with `RPLACA` or `RPLACD`. You can specify that `/RPLACA` and `/RPLACD` should be used, or `FRPLACA` and `FRPLACD`, by means of CLISP declarations (see *IRM*).

Reconstruction

You can specify a value for a pattern match operation other than what is returned by the match by writing `(MATCH` *FORM1* `WITH` *PATTERN* `=>` *FORM2*`)`.

For example,

```
(MATCH X WITH (FOO_$ 'A --) => (REVERSE FOO))
```

translates as:

```
[PROG ($$2)
   (RETURN
      (COND ((SETQ $$2 (MEMB 'A X))
             (SETQ FOO (LDIFF X $2))
             (REVERSE FOO]
```

Place markers in the pattern can be referred to from within *FORM*, e.g., the above could also have been written as

```
(MATCH X WITH (!#1 'A --) => (REVERSE #1)).
```

If -> is used in place of =>, the expression being matched is also physically changed to the value of *FORM*.

For example,

```
(MATCH X WITH (#1 'A !#2) -> (CONS #1 #2))
```

would remove the second element from X, if it were equal to A.

In general, `(MATCH` *FORM1* `WITH` *PATTERN* `->` *FORM2*`)` is translated so as to compute *FORM2* if the match is successful, and then smash its value into the first node of *FORM1*. However, whenever possible, the translation does not actually require *FORM2* to be computed in its entirety, but instead the pattern match compiler uses *FORM2* as an indication of what should be done to *FORM1*.

For example,

```
(MATCH X WITH (#1 'A !#2) -> (CONS #1 #2))
```

translates as

```
(AND (EQ (CADR X) 'A) (RPLACD X (CDDR X)))
```

Limitation

The pattern match facility does not contain some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, you can be confident that what facilities it does provide results in Lisp expressions comparable to those you would generate by hand.

Examples

```
(MATCH X WITH (-- 'A --))
```

-- matches any arbitrary segment. `'A` matches only an `A`, and the second -- again matches an arbitrary segment; thus this translates to `(MEMB 'A X)`.

```
(MATCH X WITH (-- 'A))
```

Again, -- matches an arbitrary segment; however, since there is no -- after the `'A`, `A` must be the last element of `X`. Thus this translates to: `(EQ (CAR (LAST X)) 'A)`.

```
(MATCH X WITH ('A 'B -- 'C $3 --))
```

`CAR` of `X` must be `A`, and `CADR` must be `B`, and there must be at least three elements after the first `C`, so the translation is:

```
(AND (EQ (CAR X) 'A)
     (EQ (CADR X) 'B)
     (CDDDR (MEMB 'C (CDDR X))))
```

```
(MATCH X WITH (('A 'B) 'C Y_$1 $))
```

Since `('A 'B)` does not end in `$` or `--`, `(CDDAR X)` must be `NIL`. The translation is:

```
(COND
   ((AND (EQ (CAAR X) 'A)
         (EQ (CADAR X) 'B)
         (NULL (CDDAR X))
         (EQ (CADR X) 'C)
         (CDDR X))
    (SETQ Y (CADDR X)) T))
```

```
(MATCH X WITH (#1 'A $ 'B 'C #1 $))
```

`#1` is implicitly assigned to the first element in the list. The `$` searches for the first `B` following `A`. This `B` must be followed by a `C`, and the `C` by an expression equal to the first element. The translation is:

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CADR X) 'A)
           (EQ [CADR (SETQ $$2 (MEMB 'B (CDDR X] 'C)
           (CDDR $$2)
           (EQUAL (CADDR $$2) (CAR X]
```

```
(MATCH X WITH (#1 'A -- 'B 'C #1 $))
```

Similar to the pattern above, except that `--` specifies a search for *any* `B` followed by a `C` followed by the first element, so the translation is:

```
[AND (EQ (CADR X) 'A)
     (SOME (CDDR X)
           (FUNCTION (LAMBDA ($$2 $$1)
              (AND (EQ $$2 'B)
                   (EQ (CADR $$1) 'C)
                   (CDDR $$1)
                   (EQUAL (CADDR $$1) (CAR X]
```

[This page intentionally left blank]