Skillnader mellan Elisp och andra Lispdialekter

Variabler i Elisp har dynamiskt scope:

```
(setq a 0)
(defun foo ()
  (+ a 4))

(let ((a 5))
  (foo))

=> 9
```

Detta utnyttjas ofta i Elisp:

```
(let ((case-fold-search nil))
  (search-forward "cyberpromo"))
```

Evalueringsordningen skiljer sig från Scheme. Funktionen (det första elementet) evalueras inte alls, och sedan evalueras vart och ett av argumenten från vänster till höger.

För att dynamiskt bestämma funktionen använder man funcall eller apply:

```
(apply func ’x1 ’x2 ’(x3 x4))
(funcall func ’x1 ’x2 ’x3 ’x4)
```

Buffrar

Skapa en buffer:

```
(get-buffer BUFFER-NAME)
(get-buffer-create BUFFER-NAME)
(generate-new-buffer BUFFER-NAME)
```

Aktuell buffer:

```
(set-buffer BUFFER)
(current-buffer)
```

Point pekar ut den plats som text sätts in på, sökningar startar från och annat:

```
(point)
(goto-char POS)
(point-min)
(point-max)
(forward-char &optional COUNT)
(forward-word COUNT)
(forward-line &optional COUNT)
(beginning-of-line &optional COUNT)
(end-of-line &optional COUNT)
```

För att tillfälligt ändra aktuell buffer och point kan man använda `save-excursion`:

```
(save-excursion
  (set-buffer (get-buffer-create
                "*Article*"))
  (goto-char (point-min))
  (insert "Foo-header: gazonk\n"))
```

Insättning och radering av text

Sätta in text vid point i aktuell buffer:

```
(insert &rest TEXT)
(insert-char CHAR COUNT)
(insert-buffer-substring
  FROM-BUFFER &optional START END)
```

Radera text ur aktuell buffer:

```
(erase-buffer)
(delete-region START END)
(delete-char COUNT &optional KILLP)
```

Textformattering

Funktioner som bygger strängar:

```
(make-string LENGTH INIT)
(concat &REST STRINGS)
(format FORMAT &rest OBJECTS)
```

Formatteringssträngen `FORMAT` liknar `printf` i C:

- `%s`: Ersätts med objektets textrepresentation.
- `%S`: Som %s, men den resulterande strängen kan läsas in igen och ge ett identiskt objekt.
- `%d`, `%o`, `%x`: Skriv ett heltal i decimal, oktal respektive hexadecimal bas.
- `%c`: Ersätts med det tecken som talet representerar.
- `%e`, `%f`, `%g`: Skriv ut ett flyttal i olika varianter.

Ett tal mellan `%` och typen ger fältvidden, och texten högerjusteras inom det. Ett negativ tal ger vänsterjustering.

```
(format "%s %d %c" 65 65 65)

=> "65 65 A"

(format "%s %S" "foo" "foo")

=> "foo \"foo\""

(format "%S" '(a ("b" ?c)))

=> "(a (\"b\" 99))"
```

Sökning och matchning

Exakt sökning:

```
(search-forward STRING
  &optional LIMIT NOERROR REPEAT)
(search-backward STRING
  &optional LIMIT NOERROR REPEAT)
(word-search-forward STRING
  &optional LIMIT NOERROR REPEAT)
```

Regexpsökning:

```
(re-search-forward REGEXP
  &optional LIMIT NOERROR REPEAT)
(re-search-backward REGEXP
  &optional LIMIT NOERROR REPEAT)
```

Om sökningen lyckas flyttas point till slutet (eller början) av det matchande, och det nya värdet på point returneras.
Om sökningen misslyckas signaleras ett fel om `NOERROR` är `nil`, om den är `t` returneras `nil` och point flyttas inte, och om den är något annat returneras `nil` och point flyttas till `LIMIT`.

Om `case-fold-search` är `t` anses stora och små bokstäver vara lika.

Annan matchning:

```
(string-match REGEXP STRING
  &optional START)
(looking-at REGEXP)
```

Regexpsyntax

`. * + ? [...] [^...] ^ $` fungerar som vanligt.
Elisputbyggnader:

`\|` Specifierar alternativa regexps.

`\(...\)`
Grupperar en regexp, har tre användsområden:
1. Avgänsa `\|` alternativ.
2. Låta `*`, `+` samt `?` arbeta på hela uttrycket inom paranteser.
3. Komma ihåg en matchad delsträng för senare åtkomst.

`\DIGIT`
Matchar samma text som matchades av grupp `DIGIT` (1-9).

`` \` `` Matchar början av buffern eller strängen.

`\'` Matchar slutet av buffern eller strängen.

`\=` Matchar vid point.

`\b` Matchar början eller slutet av ett ord.

`\B` Motsatsen till `\b`.

`\<` Matchar början av ett ord.

`\>` Matchar slutet av ett ord.

Matchdata

De delsträngar som matchades av grupper i en regexpmatchning kan kommas åt:

```
(match-beginning COUNT)
(match-end COUNT)
(match-string COUNT &optional STRING)

(match-data)
(store-match-data MATCH-LIST)
(save-match-data &rest BODY)
```

Byta ut den matchade texten:

```
(replace-match REPLACEMENT
   &optional FIXEDCASE LITERAL)
```

Om `LITERAL` är `t` kan följande användas i `REPLACEMENT`:

- `\&` Hela texten som ersätts.
- `\N` Delsträngen som matchade den N:te gruppen.
- `\\` Ett `\`.

Markörer

En markör pekar ut en position i en buffer. När text sätts in före markören flyttas den framåt motsvarande sträcka, och när text raderas flyttas den bakåt.

```
(make-marker)
(point-marker)
(copy-marker MARKER-OR-INTEGER)

(markerp)
(integer-or-marker-p)
(number-or-marker-p)
(marker-position MARKER)
(marker-buffer MARKER)

(set-marker MARKER POSITION
   &optional BUFFER)

(insert-before-markers &rest TEXT)
```

Varje buffer har en speciell markör: mark.

```
(mark &optional FORCE)
(mark-marker)
(set-mark POSITION)
```

Bufferlokala variabler

Variabler kan vara lokala till en viss buffer. När den buffern är aktuell syns värdena på de variablerna.

```
(make-local-variable VARIABLE)
(make-variable-buffer-local VARIABLE)
(setq-default SYMBOL VALUE)
(kill-local-variable VARIABLE)
(kill-all-local-variables)
(buffer-local-variables
    &optional BUFFER)
```

Kommandon

Ett kommando är en funktion med en `interactive`-form:

```
(defun te ()
  "Pip när teet är klart."
  (interactive)
  (run-at-time 270 nil 'te-pip))

(defun te-pip ()
  (ding) (ding)
  (message "Teet är klart!")
```

Nu kan man köra kommandot `te` med M-x, men inte funktionen `te-pip`.

Om kommandot ska ta argument kan `interactive` beräkna dem:

```
(defun te2 (mins)
  "Pip när teet är klart, efter MINS
minuter."
  (interactive "p")
  (run-at-time (* mins 60) nil
               'te-pip))
```

Strängen i `interactive` talar om varifrån argumenten ska komma, `p` säger att prefixargumentet (C-u) ska användas. Det finns många fler tecken som kan användas, vissa använder minibuffern för att be om argumentet.

Kommandon (fortsätting)

Om argumentet till `interactive` inte är en sträng evalueras det. Resultatet ska vara en lista, som ger värdet på motsvarande argument:

```
(defun te3 (mins)
  "Pip när teet är klart, eller efter
MINS minuter. Om MINS inte ges som
prefix används 5 minuter."
  (interactive
   (list (if current-prefix-arg
             (prefix-numeric-value
              current-prefix-arg)
           5)))
  (run-at-time (* mins 60) nil
               'te-pip))
```

Keymaps

En keymap binder kommandon till tangenter.

```
(make-keymap)
(make-sparse-keymap)
(copy-keymap KEYMAP)

(current-global-map)
(current-local-map)
(use-global-map KEYMAP)
(use-local-map KEYMAP)

(define-key KEYMAP KEY BINDING)
```

KEY är en sträng eller en vektor med symboler och tecken:

Sekvens	Sträng	Vektor
`f`	`"f"`	`[?f]`
`M-f`	`"\M-f"`	`[?\M-f]`
`C-x C-f`	`"\C-x\C-f"`	`[?\C-x ?\C-f]`
`Left`		`[left]`
`Control-Right`		`[C-right]`
`Mouse-1`		`[mouse-1]`
`Shift-mouse-3`		`[S-mouse-3]`

I XEmacs heter `mouse-N` istället `button-N`.

Major Modes

```
;; mitt-mode.el -- Mitt alldeles egna mode

(defvar mitt-mode-map nil
  "Keymap till mitt-mode.")

(if mitt-mode-map
    nil
  (setq mitt-mode-map (make-sparse-keymap))
  (define-key mitt-mode-map "f" 'foo-function))

(defvar mitt-mode-hook nil
  "Hooks till mitt-mode.")

(defvar mitt-mode-counter nil
  "Räknare för foo-function.")

(defun mitt-mode ()
  "Mitt eget mode.
\\{mitt-mode-map}"
  (interactive)
  (setq major-mode 'mitt-mode
        mode-name "Mitt eget mode")
  (use-local-map mitt-mode-map)
  ;; (kill-all-local-variables)
  (set (make-local-variable 'mitt-mode-counter) 0)
  (run-hooks mitt-mode-hook))

(defun foo-function ()
  "Räknar upp i mitt-mode."
  (interactive)
  (message "Foos: %d" mitt-mode-counter)
  (setq mitt-mode-counter (1+ mitt-mode-counter)))

(provide 'mitt-mode)
```

Minor modes

```
;; mitt-minor-mode.el -- Mitt egna minor mode

(defvar mitt-minor-mode-map nil
  "Keymap till mitt-mode.")

(if mitt-minor-mode-map
    nil
  (setq mitt-minor-mode-map (make-sparse-keymap))
  (define-key mitt-minor-mode-map "f"
              'foo-minor-function))

(defvar mitt-minor-mode-hook nil
  "Hooks till mitt-minor-mode.")

(defvar mitt-minor-mode-counter nil
  "Räknare för foo-function.")

(defvar mitt-minor-mode nil
  "Mitt eget minor mode.")

(defun mitt-minor-mode (&optional arg)
  "Mitt eget mode.
\\{mitt-minor-mode-map}"
  (interactive "P")
  (setq mitt-minor-mode
        (if (null arg) (not mitt-minor-mode)
            (> (prefix-numeric-value arg) 0)))
  (if mitt-minor-mode
      (set (make-local-variable
            'mitt-minor-mode-counter) 0)
    (kill-local-variable
     'mitt-minor-mode-counter)))
```

Minor Modes (forts.)

```
(or (assq 'mitt-minor-mode minor-mode-alist)
    (setq minor-mode-alist
         (cons '(mitt-minor-mode " MittMinorMode")
         minor-mode-alist)))

(or (assq 'mitt-minor-mode minor-mode-map-alist)
    (setq minor-mode-map-alist
         (cons (cons 'mitt-minor-mode
                     mitt-minor-mode-map)
               minor-mode-map-alist)))

(defun foo-minor-function ()
  "Räknar upp i mitt-minor-mode."
  (interactive)
  (message "Foos: %d" mitt-minor-mode-counter)
  (setq mitt-minor-mode-counter
        (1+ mitt-minor-mode-counter)))

(provide 'mitt-minor-mode)
```

Ärvt mode

Om ett mode nästan gör vad man behöver kan man skapa ett nytt som ärver det gamla modets egenskaper:

```
(define-derived-mode PARENT CHILD NAME
  &optional DOCSTRING &rest BODY)
```

Om man vill använda minor-mode från tidigare, men vill köra foo-function då man trycker M-f istället för f kan man göra så här:

```
(define-derived-mode mitt-mode mitt-nya-mod
  "Mitt nya mode"
  "Mitt nya mode
\\{mitt-nya-mode-map}"
  (define-key mitt-nya-mode-map "f" nil)
  (define-key mitt-nya-mode-map "\M-f"
              'foo-function))
```