Struktury

Struktura jest to typ danych który zawiera pola którym można przypisać odpowiednią wartość.

Aby użyć struktury należy ją najpierw zadeklarować, za pomocą makra `defstruct`.

```
(defstruct struct p1 p2 p3)
```

Pierwszy parametr `defstruct` to nazwa struktury (struct), a pola p1, p2, p3 są nazywane slotami jeśli tworzą pary nazwa-wartość.

Makro `defstruct` automatycznie utworzy funkcję w postaci make-<nazwa struktury> w naszym przypadku będzie to funkcja `make-struct` tworząca nowy obiekt strukturalny (rekord) oraz funkcje odwołujące się do poszczególnych pól `struct-p1, struct-p2 i struct-p3.`

```
(setq s1 (make-struct :p1 1 :p2 2 :p3 3))
(setf (struct-p1 s1) "Lorem Ipsum")
(print (struct-p1 s1))
(incf (struct-p2 s1))
```

Powyższy kod tworzy jeden rekord o nazwie `s1` następnie przypisuje polu `p1` wartość "Lorem Ipsum", następnie wyświetla jego zawartość. Jeśli nie przypiszemy wartości pola przy tworzeniu rekordu, będzie on miał wartość `nil`.

Przykład definicji struktury osoba i operacji na slotach:

```
(defstruct osoba imie nazwisko wiek plec)

(setq pracownik1 (make-osoba :imie "Jan" :nazwisko "Kowalski" :wiek 30
:plec "M"))

(print pracownik1)

(print (osoba-imie pracownik1))
```

Istnieje możliwość zastosowania wartości domyślnych dla pól aby nie trzeba było określać ich przy tworzeniu rekordu.

```
(defstruct nazwa_struktury
        (slot1 wartość_domyślna)
        (slot2 wartość_domyślna)
        …
        (slotn wartość_domyślna)
)
```

Wartości domyślne są opcjonalne.

```
(defstruct osoba (imie "brak danych") nazwisko (wiek 0)(plec "M"))
(setq a1 (make-osoba))
```

Istnieje możliwość skopiowania obiektu struktury za pomocą funkcji `copy-nazwa_struktury`, przykład:

```
(setq a2 (copy-osoba a1))
```

Sprawdź następujące wyrażenia:

1. `(eq a1 a2)`
2. `(eq (osoba-imie a1) (osoba-imie a2))`
3. `(osoba-p a1)`
4. `(osoba-p "Ktos")`
5.

Deklaracja pól tylko do odczytu:

```
(defstruct osoba2 (imie "Brak" :read-only t) nazwisko wiek)
(setq x2 (make-osoba2))
(setf (osoba2-nazwisko x2) "Kowalski")
(setf (osoba2-wiek x2) 20)
```

Poniższa operacja nie może zostać wykonana:

```
(setf (osoba2-imie x2) "Jan")
```

Istnieje możliwość tworzenia rekordów bez użycia parametrów kluczowych za pomocą zdefiniowanego konstruktora.

```
(defstruct (struktura
            (:constructor
             create-struktura (pole1 pole2 pole3)))
  pole1 pole2 pole3)

(setq x4 (create-struktura "Lorem" "Ipsum" "Dolor"))
(print (struktura-pole1 x4))
```

Istnieje możliwość dziedziczenia tzn. że pola z struktury, która jest potomkiem zostaną przekazane od rodzica.

```
(defstruct (struktura2
            (:include struktura))
  pole4)

(setq x5 (make-struktura2 :pole1 "Lorem" :pole4 "Amet"))
(print (struktura2-pole1 x5))
=> "Lorem"
(print (struktura2-pole4 x5))
=> "Amet"
```

Klasy i obiekty

Aby utworzyć nowy typ obiektu czyli klasę należy użyć makra `defclass`.

```
(defclass klasa ()
  (a b c))
```

Aby znaleźć obiekt klasy należy użyć funkcji `find-class`.

```
(find-class 'klasa)
=> #<STANDARD-CLASS KLASA>
```

Aby utworzyć nowy obiekt danej klasy należy użyć funkcji `make-instance`.

```
(setq obj (make-instance 'klasa))
```

Aby odwołać się do poszczególnych pól obiektu należy użyć funkcji `slot-value`.

```
(setf (slot-value obj 'a) "Lorem")
(setf (slot-value obj 'b) "Ipsum")
(setf (slot-value obj 'c) "Dolor")
(print (slot-value obj 'a))
=> "Lorem"
```

Jeśli nastąpi odwołanie do pola za pomocą funkcji `slot-value`, zanim zostanie przypisana do niego wartość Common Lisp zgłosi błąd.

Zamiast używać funkcji `slot-value` można użyć specjalnej funkcji dostępowej za pomocą parametru kluczowego `:accessor`.

```
(defclass klasa ()
  ((wartosc :accessor klasa-wartosc)))
(setq obj (make-instance 'klasa))
(setf (klasa-wartosc obj) "Lorem Ipsum")
(print (klasa-wartosc obj))
=> "Lorem Ipsum"
```

Można także użyć osobnej funkcji do zapisu i odczytu wartości pola za pomocą odpowiednio `:writer` i `:reader`.

```
(defclass klasa ()
  ((wartosc :reader klasa-get-wartosc :writer klasa-set-wartosc)))

(setq obj (make-instance 'klasa))
```

```
(klasa-set-wartosc "Lorem Ipsum" obj)
(print (klasa-get-wartosc obj))
=> "Lorem Ipsum"
```

Dodatkowymi opcjami dla pól są: `:documentation` określające dokumentacje pola. `:type` określająca typ pola (może być jeden z: `real, integer, fixnum, float, bignum, ratinal lub complex`) `:initform` określa wartość domyślną dla pola klasy.

Istnieje także możliwość określenia pól statycznych tzn. takich, dla których wartość jest przypisywana dla klasy, wartość dla każdego obiektu będzie taka sama za pomocą parametru kluczowego `:allocation` z przypisaną wartością `:class`.

Dziedziczenie

Tak jak w przypadku struktur klasy mogą dziedziczyć pola po swoich rodzicach.

```
(defclass klasa ()
  ((a :accessor klasa-a)
   (b :accessor klasa-b)
   (c :accessor klasa-c)))

(defclass klasa2 (klasa)
  ((d :accessor klasa2-d :initarg :d)))

(setq obj (make-instance 'klasa2 :d "Lorem Ipsum"))
(setf (klasa-a obj) "Dolor")
(print (klasa-a obj))
=> "Dolor"
```

W powyższym kodzie użyto opcji dla pola o nazwie `:initarg`, dzięki której jest możliwość przypisania polu wartości przy konstruowaniu obiektu za pomocą funkcji `make-instance`.

Można także użyć opcji dla pola `:initform` określającą wartość domyślną dla danego pola.

Istnieje możliwość dziedziczenia po kilku klasach tzw. dziedziczenie wielokrotne.

Funkcje ogólne (generic functions)

Funkcje ogólne definiuje się podobnie tak jak zwykłe funkcje za pomocą makra `defgeneric`. Funkcje ogólne nie posiadają kodu (ciała). Funkcje ogólne są podobne do funkcji wirtualnych z języka C++.

```
(defgeneric metoda (parametr1 parametr2))
```

Funkcji ogólnej opcjonalnie można przypisać dokumentacje.

Metody

Metody definiuje się za pomocą makra `defmethod`.

```
(defmethod metoda ((self klasa2) (value string))
  (setf (slot-value self 'd) value))

(setq obj (make-instance 'klasa2))
(metoda obj "Dolor Sit Amet")
(print (klasa2-d obj))
=> "Dolor Sit Amet"
```

Powyższa metoda przypisuje wartość pola `quux` klasa `klasa2`.

Metody mogą być "przypisane" do kilku klas na raz, jak w powyższym przypadku do klasy `klasa2` i klasy `string`.

Metody `:before` i `:after` są wywoływane odpowiednio przed i wywołaniu metody. Funkcja `:around` może być wywoływana zamiast metody, jest wywoływana przed wszystkimi metodami.

```
(defmethod metoda :before ((self klasa2) (value string))
  (print "wywołanie :before metoda"))
(defmethod metoda :after ((self klasa2) (value string))
  (print "wywołanie :after metoda"))
(defmethod metoda :around ((self klasa2) (value string))
  (print "wywołanie :around metoda")
  (call-next-method self value))

(metoda obj "a b")
"wywołanie :around metoda"
"wywołanie :before metoda"
"wywołanie :after metoda"
```

Funkcja `:around` musi wywoływać funkcję `call-next-method` aby wywołać następną metodę.

Aby usunąć metodę należy użyć poniższego kodu.

```
(remove-method #'metoda (find-method #'metoda ()
                                     (list (find-class 'klasa2)
                                           (find-class 'string))))
```

Pierwszym argumentem `remove-method` jest funkcja ogólna a drugim metoda.

Funkcje

Funkcje definiujemy za pomocą makra `defun`. Funkcje istnieją w oddzielnej przestrzeni nazw, dzięki czemu możemy tworzyć zmienne i funkcje o takich samych nazwach.

```
(defun nazwa (parametr1 parametr2)
  "Dokumentacja dla funkcji."
  wyrażenie
  ...)
; wywołanie
(nazwa 1 2)
```

Funkcje mogą mieć parametry opcjonalne:

```
(defun nazwa (parametr &optional (param2 default param2-p))
  wyrażenie
  ..)

(defun test (a &optinal (b 0))
  (print a)(print b)
)

(test 1)
1
0
0
(test 1 2)
1
2
2
```

Wartość `default`, w parametrze opcjonalnym, określa wartość jaka zostanie przypisana jeśli funkcja zostanie wywołana z jednym parametrem. `param2-p` określa czy parametr został przypisany.

Istnieje możliwość aby funkcje przyjmowały różną liczbę argumentów, za pomocą `&rest`.

```
(defun test (a &rest rest)
  (format t "a:~a rest:~a~%" a rest))

(test 1)
"a:1 rest:NIL"
(test 1 2)
"a:1 rest:(2)"
(test 1 2 3)
"a:1 rest:(2 3)"
```

Aby określić parametry kluczowe dla danej funkcji używamy parametru `&key`.

```
(defun test (a &key b (c 10 c-p))
  (format t "a:~a b:~a c:~a c-p:~a" a b c c-p))

(test 1)
"a:1 b:nil c:10 c-p:NIL"
(test 1 :b 20)
"a:1 b:20 c:10 c-p:NIL"
(test 1 :b 20 :c 30)
"a:1 b :20 c:30 c-p:T"
```

Zmienna c-p określa czy parametr został przypisany.

Funkcje lokalne

Funkcje lokalne tworzymy za pomocą wyrażenia `flet`.

```
(flet ((nazwa1 () wyrażenie ...)
       (nazwa2 (param) wyrażenie ...))
  wyrażenie
  ...)
```

Aby zastosować wywołania rekurencyjne należy użyć wyrażenia `labels`.

```
(labels ((nazwa1 () wyrażenie ...)
         (nazwa2 (param) wyrażenie ...))
  wyrażenie
  ...)

(labels ((fact2x (x) (fact (* 2 x)))
         (fact (x) (if (< x 2) 1 (* x (fact (1- x))))))
  (fact2x 3))


(labels ((foo () 2)
         (bar () (foo)))
  (bar))
```

W powyższym kodzie funkcja `foo` zwraca wartość 2 funkcja `bar` wywołuje funkcję `foo`. Wewnątrz funkcji `labels` wywoływana jest funkcja `bar`.

Rekurencja

Funkcja rekurencyjna to taka funkcja, w której wnętrzu następuje odwołanie do niej samej tzw. wywołanie rekurencyjne.

Poniżej przedstawiono funkcje rekurencyjną obliczającą silnię.

```
(defun factorial (n)
  (if (zerop n) 1
    (* n (factorial (- n 1)))))
```

Rekurencja ogonowa

Jeżeli w funkcji wywołanie rekurencyjne występuje na końcu wyrażenia, to taka funkcja wykorzystuje tzw. rekurencje ogonową (ang. tail recusion), takie wywołanie rekurencyjne zapisane zostanie jak iteracja za pomocą instrukcji skoku, dzięki czemu funkcja nie będzie wykorzystywać stosu (stanie się bardziej efektywna).

Poniżej przedstawiono funkcję obliczającą silnie z rekurencją ogonową.

```
(defun factorial (n)
  (labels ((f (n product)
             (if (zerop n) product
               (f (- n 1) (* product n))))) ; wywołanie rekurencyjne
ogonowe
          (f n 1)))
```

Funkcje anonimowe

Funkcje anonimowe zapisujemy za pomocą lambda abstrakcji.

```
(lambda (x)
  (* x x))
```

Aby wywołać funkcje anonimową używamy funkcji funcall lub apply.

```
(funcall (lambda (x) (* x x)) 10)
=> 100

(setq square (lambda (x) (* x x)))
(funcall square 20)
=> 400

(apply #'+ (list 1 2 3 4))
=> 10
```

Funkcja apply przyjmuje jako drugi argument listę parametrów. Przekazując funkcję jako parametr do innej funkcji należy ją cytować za pomocą znaków #'.

Funkcje wyższego rzędu

Funkcje mogą być przekazywane jako parametry, mogą też być zwracane przez funkcję.

```
(defun complement (fun)
```

```
  (lambda (x)
    (not (funcall fun x))))
```

Funkcja ta jest zdefiniowana w standardzie Common Lisp.

Aby przekazać funkcję jako parametr należy użyć cytowania funkcji - znaków przed nazwą funkcji #'.

```
(mapcar #'sqrt '(1 2 3 4))
```

Funkcje mapujące

Funkcje mapujących używamy jeśli chcemy iterować po jakiejś liście lub sekwencji i przypisać do każdego elementu wartość zwracaną przez funkcje.

```
(defun square (x)
  (* x x))

(mapcar #'square '(1 2 3 4 5 6))
=> (1 4 9 16 25 36)

(mapcar (lambda (x) (* x x)) '(1 2 3 4 5))
=> (1 4 9 16 25)
```

Funkcje mapujące:

`mapcar`

```
(mapcar #'funkcja lista1 lista2 ...)
```

Funkcja przekazywana jako drugi parametr musi mieć tyle argumentów ile przekazano list.

`mapcan`

```
(mapcan #'funkcja lista2 lista2 ...)
```

```
(mapcan (lambda (x) (list (+ x 10) 'x)) '(1 2 3 4))

(mapcan #'list '(a b c d))

mapcan (lambda (x) (if (> x 0) (list x) nil)) '(-4 6 -23 1 0 12 ))
```

Funkcja działa tak jak mapcar z tym że parametr `typ` może przyjmować wartości np.: list, array, vector, string.

`maplist`

```
(maplist #'funkcja lista1 lista2 ...)
```

```
(maplist (lambda (x) (list 'start x 'end)) '(1 2 3 4))
```

Funkcja działa tak jak `mapcar` z tym że do funkcji przekazywane są pary zamiast elementów list.

Domknięcia leksykalne

```
(let ((x 0))
  (defun foo ()
    (setq x (incf x))
    x))

(foo)
=> 1
(foo)
=> 2
(foo)
=> 3
```

W powyższej funkcji zmienna x jest zamknięta wewnątrz funkcji `foo` (funkcja może korzystać ze zmiennej x) chociaż zmienna jest już niedostępna poza konstrukcją `let`.

Makra

Makra w odróżnieniu od funkcji nie ewaluują swoich argumentów, zamiast tego do makra przekazywane jest całe s-wyrażenia które następnie zostaną przetworzone w celu utworzenia innego s-wyrażenia. Przy wywoływaniu makra początkowe wyrażenie przekazywane jest do makra, które po przetworzeniu zwraca nowe wyrażenie. Takie wyrażenie następnie zostaje wywołane. Makro definiujemy za pomocą makra `defmacro`. W odróżnieniu od funkcji makr nie możemy przekazywać jako parametru do innych funkcji.

```
(defmacro def (args &body body)
  `(defun ,args ,@body))
```

Powyższy kod tworzy makro tworzące nową funkcje, używa ono znacznika `&body` dla określenia ciała definiowanej funkcji. Powyższe makro używa składni tylnego apostrofu (backquote) `. Tylny apostrof działa podobnie do normalnego apostrofu, który cytuje swój argument. Wszystkie wyrażenia poprzedzone przecinkiem zostaną wywołane, z wyrażeń poprzedzonych znakiem przecinka i et `,@` zostaną usunięte nawiasy np.: `` `(print 1 2 3 ,(+ 1 2 (* 3 4))) `` zostanie ewoluowane przez interpreter lispa jako `(PRINT 1 2 3 15)`.

Poniżej przedstawiono makro definiujące pętlę while znaną z innych języków programowania.

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
```

```
       ,@body))

; użycie powyższego makra
(let ((x 0))
  (while (< x 10)
    (print x)
    (setq x (1+ x))))

0
1
2
3
4
5
6
7
8
9
=> NIL
```

Poniższa makro definiuje pętle for.

```
(defmacro for ((var start stop &optional return-value) &body body)
  (let ((gstop (gensym)))
    `(progn
      (do ((,var ,start (1+ ,var))
           (,gstop ,stop))
        ((> ,var ,gstop) ,return-value)
       ,@body))))

(for (i 10 20) (print i))
1
2
3
4
5
6
7
8
9
10
=> NIL

(reverse (let (result)
           (for (i 1 10 result)
                (push i result))))
=> (1 2 3 4 5 6 7 8 9 10)
```

Powyższe makro używa funkcji `gensym`, która generuje unikalną nazwę dla zmiennej, stosuje się to z powodu tego, że wewnątrz pętli `for` użytkownik może zdefiniować taką samą zmienną jaką użyliśmy wewnątrz makra, co doprowadziłoby do błędnego działania. Znacznik `&optional` definiuje argument opcjonalny tak jak w przypadku funkcji.

Za pomocą funkcji `macroexpand` można zobaczyć jak wygląda wygenerowany przez makro kod. Jest to przydatne w czasie debugowania makr. Funkcja `macroexpand` generuje kod

rekurencyjnie aż do napotkania tylko funkcji. Aby wyświetlić tylko jeden poziom makra należy użyć funkcji `macroexpand-1`.

```
(macroexpand '(for (i 10 20) (print i)))
(BLOCK NIL
 (LET ((I 10) (#:G8442 20))
  (TAGBODY #:G8443 (IF (> I #:G8442) (GO #:G8444)) (PRINT I)
   (PSETQ I (1+ I)) (GO #:G8443) #:G8444 (RETURN-FROM NIL (PROGN)))))
=> T
(macroexpand-1 '(for (i 10 20) (print i)))
(DO ((I 10 (1+ I)) (#:G8445 20)) ((> I #:G8445)) (PRINT I))
=> T
```

Nazwa zmiennej `#:G8445` została wygenerowana przez funkcje `gensym`.

Formatowanie tekstu - funkcja `format`

Funkcja `format` działa podobnie do funkcji `printf` języka C. Pierwszy parametr przyjmuje wartość `nil, t` lub strumień, jeśli zostanie przekazana wartość `nil` wynikowy ciąg znaków zostanie zwrócony przez funkcje `format` jeśli przekazana zostanie wartość `t` wynikowy ciąg znaków zostanie wypisany na standardowe wyjście, może być także przekazany strumień wyjściowy. Drugim parametr musi być ciąg znaków zawierający odpowiednie znaczniki (poprzedzone znakiem tyldy ~). Kolejne parametry zostaną wstawione w odpowiednie miejsca w ciągu znaków.

Poniżej przedstawiono listę znaczników (wielkość liter nie ma znaczenia):

`~C` - wyświetla znak

`~R` - słowna reprezentacja liczby (w języku angielskim)

`~@R` - słowna reprezentacja rzymska

`~A` - reprezentacja jako ciąg znaków

`~S` - reprezentacja jako ciąg znaków który może być wczytany za pomocą funkcji `read`

`~(znaczniki~)` - wyświetla parametr małymi literami

`~:@(znaczniki~)` - wyświetla parametr dużymi literami

`~@(znaczniki~)` - pierwszy znak zostanie wyświetlony dużymi literami

`~:(znaczniki~)` - każde słowo wielką literą

`~%` - nowa linia

`~5%` - 5 nowych linii

`~~` - znak tyldy

`~D` - liczba dziesiętna

`~10D` - liczba dziesiętna wyrównana do 10

`~X` - liczba szesnastkowa

`~O` - liczba ósemkowa

`~B` - liczba binarna

`~F` - liczba float z przecinkiem

`~,5F` - liczba float z 5 miejscami po przecinku

`~{znaczkiki~}` - iteracja - parametr musi być listą

```
(format t "~{~a ~}~%" (list 1 2 3 4 5))
1 2 3 4 5
=> NIL
```

`~@{znaczniki~}` - iteracja - przetwarza parametry jak listę

```
(format t "~{~a ~}~%" 1 2 3 4 5)
1 2 3 4 5
=> NIL
```

`~[forma1~;forma2~;..~;N~]` - parametr wybiera formę

```
(format t "~[zero~;jeden~;dwa~]~%" 1)
"jeden"
=> NIL
```

`~:[forma-fałsz~;forma-prawda~]` - wybiera formę w zależności czy parametr przyjmuje wartość `t` lub `nil`

```
(let ((x 10))
  (format nil "~:[fałsz~;prawda~]" (zerop x)))
=> "fałsz"
```

`~*` - omija jeden parametr

`~:*` - cofa się o jeden parametr (przetwarza jeden parametr dwa razy)

`V` - wstawia wartość parametru

```
(format nil "~V%" 5)
=> "


" ; 5 wierszy
```

Strumienie - obsługa wejścia/wyjścia

Aby otworzyć plik należy użyć funkcji `open`.

```
(setq uchwyt (open "nazwa-pliku" :direction :output))
```

parametr kluczowy `:direction` może przyjmować jedną z dwóch wartości `:input` lub `output` dla których funkcja odpowiednio otwiera plik wejściowy (z którego można czytać) i wyjściowy (do którego można zapisywać).

Funkcji `open` można przekazać parametr kluczowy `:if-exist` któremu można przekazać wartość `:supersede` - zastępującą zawartość, `:append` - dodająca zawartość na końcu pliku. Za pomocą funkcji `read`, `read-line`, `read-char` i `write-byte` można odpowiednio przeczytać wyrażenie lispowe ze strumienia, przeczytać linie, przeczytać pojedynczy znak oraz przeczytać bajt danych. Funkcje przyjmują jako drugi opcjonalny parametr strumień wejściowy.

Analogicznie za pomocą funkcji `write`, `write-line`, `write-char` i `write-byte` można zapisać wyrażenie do strumienia, zapisać linie, zapisać znak oraz zapisać bajt danych. Funkcja przyjmuje jako drugi opcjonalny parametr strumień wyjściowy. Można użyć także funkcji `print`.

```
(let ((file (open "plik" :direction :input)))
  (write-line (read-line file))
  (close file))
```

Po zakończeniu przetwarzania strumienia należy wywołać funkcję `close` która zamyka strumień. Strumienie mogą być buforowane, dlatego zapis na dysk może nastąpić dopiero po zamknięciu strumienia.

Funkcja `listen` sprawdza czy jest jakiś znak do odczytania ze strumienia.

Makro `with-open-file` zamyka plik automatycznie (także gdy nastąpi błąd).

```
(with-open-file (file "plik" :direction :input)
  (write-line (read-line file)))
```

Poniżej przedstawiono funkcję która zwraca listę linii z pliku.

```
(defun read-lines (filename)
  "Function read file and return list of lines."
  (with-open-file (file filename :direction :input)
                (loop while (listen file)
                      collect (read-line file))))
```

Funkcja `file-length` zwraca wielkość pliku.

Funkcja `delete-file` usuwa plik.

Funkcja `rename-file` zmienia nazwę pliku.

```
(rename-file "old-name" "new-name")
```

Strumienie w pamięci.

Za pomocą makra `with-input-from-string` można utworzyć strumień z ciągu znaków.

```
(with-input-from-string (stream "\"Lorem Ipsum Dolor Sit Amet\"")
                        (write (read stream)))
"Lorem Ipsum Dolor Sit Amet"
=> "Lorem Ipsum Dolor Sit Amet"
```

Za pomocą makra `with-output-to-string` można zapisywać łańcuchy znaków do strumienia.

```
(with-output-to-string (stream)
                        (princ "Lorem Ipsum Dolor Sit Amet" stream))
=> "Lorem Ipsum Dolor Sit Amet"
```

Obsługa błędów

Za pomocą funkcji `unwind-protect` można wywołać określony kod nawet w przypadku jeśli wystąpił błąd.

```
(let (file)
  (unwind-protect
      (progn
        (setq file (open "file" :direction :input))
        (loop while (listen file) do
              (write-line (read-line file))))
    (when file (close file))))
```

Makro `with-open-file` jest zdefiniowane właśnie za pomocą `unwind-protect`.

Błąd można prosto zasygnalizować za pomocą funkcji `error`. Błąd w programie przerywa jego działanie. Jeśli korzystamy z interpretera Lispu w czasie wystąpienia błędu nastąpi wejście do Debugera.

```
(error "Wystąpił błąd w programie!")
```

Istnieje także możliwość definiowania własnych błędów (ang. conditions) za pomocą makra `define-condition`. Błędy działają podobnie do klas, istnieje możliwość dziedziczenia. Wszystkie definiowane błędy powinny dziedziczyć po `error`. Błędy działają podobnie do wyjątków w takich językach jak C++, Java czy Python.

```
(define-condition Foo(error)
  ((co :initarg :co :initform "Błąd Foo" :reader co))
  (:report (lambda (condition stream)
             (format stream "Nastąpił błąd: ~@(~A~)." (co condition))))
  (:documentation "Podstawowy Błąd Foo"))
```

W powyższym kodzie użyto znacznika `:report` na określenie funkcji za pomocą której zostanie wyświetlona informacja o błędzie. Funkcja ta przyjmuje dwa argumenty - obiekt błędu oraz strumień. Znacznik `:documentation` określa dokumentacje do błędu. Pola w błędach definiujemy tak jak w przypadku klas.

Aby wywołać błąd należy użyć funkcji `error` tak jak w prostym przypadku z ciągiem znaków.

```
(error 'Foo :co "Jakiś błąd w programie")
*** - Nastąpił błąd: Jakiś błąd w programie.
```

Aby użyć dziedziczenia błędów należy zastosować poniższy kod.

```
(define-condition Bar(Foo)
  ((dlaczego :initarg :dlaczego :reader dlaczego))
  (:report (lambda (condition stream)
             (format stream "Error: ~(~A~) jest błędny. Dlaczego? ~@(~A~)."
                     (co condition)
                     (dlaczego condition)))))

(error 'Bar :co "poważny błąd" :dlaczego "nieznany")
*** - Error: poważny błąd jest błędny. Dlaczego? nieznany.
```

Za pomocą makra `ignore-errors` można ignorować błędy np:

```
(ignore-errors (error "Ten błąd zostanie zignorowany."))
=> NIL
```

Pakietowanie

Programy napisane w Common Lispie można zapisywać w plikach. Aby wczytać plik należy użyć funkcji `load`.

```
(load "nazwa_pliku.lisp")
```

Aby utworzyć nowy pakiet należy użyć makra `defpackage`.

```
(defpackage :Pakiet
  (:documentation "To jest dokumentacja tego Pakietu.")
  (:use :common-lisp)
  (:export :range :factorial))

(provide "Pakiet")

(defun factorial (n)
  "Return factorial of n (n!)."
  (labels ((f (n product)
              (if (zerop n) product
                (f (- n 1) (* product n)))))
          (f n 1)))

(defun range (n)
  "return list of n integers."
  (let (result)
    (dotimes (i n)
      (push i result))
    (nreverse result)))
```

Powyższy pakiet definiuje dwie funkcje `factorial` oraz `range` które są zadeklarowane w pakiecie za pomocą parametru kluczowego `:export`. Znacznik `:use` określa z których pakietów będziemy korzystać. Znacznik `:documentation` określa dokumentacje pakietu. Powyższy kod należy umieścić w pliku `Pakiet.lisp`

Aby móc używać tak zdefiniowanego pakietu należy użyć funkcji `require`, należy także określić za pomocą funkcji `in-package`, że będziemy korzystać z danego pakietu. W przypadku nie wywołania funkcji `in-package` musielibyśmy używać pełnej nazwy każdej z funkcji np. `Pakiet:factorial`.

```
(require :Pakiet)
(in-package :Pakiet)

(factorial 10)
=> 3628800

(reduce #'* (mapcar #'1+ (range 10)))
=> 3628800
```

Zamiast funkcji `require` moglibyśmy także użyć funkcji load.