

Structures in LISP

✓ Lists are powerful data structures, but...

- ✗ you may have to define your own functions for accessing meaningful elements of the data structure

Example: representing facts in a database as 3-element lists

(object attribute value)

```
(defun fact-object (fact)
  (if (or (not (listp fact))
          (not (= (length fact) 3)))
      (error)
      (first fact)))
```

```
(defun fact-attribute (fact)
```

...

```
(defun fact-value (fact)
```

...

- ✗ you may have to use tricks to distinguish lists representing different data structures with the same list structure

Example: distinguishing assertions in a genealogical database represented as 3-element lists

(person father mother)

from facts in the blocks-world database

(object attribute value)

Structures in LISP

- ✓ Structure types are user-defined data types in LISP
- ✓ Structures come with automatically-defined element access procedures
- ✓ Structures come with automatically-defined type predicates
- ✓ Structures permit defining default values for structure elements
- ✓ Structures support data abstraction: hiding low-level details of implementation of a data structure, permitting the programmer to concentrate on high-level concepts

Defining structure types

- ✓ `defstruct` macro defines structure types

```
(defstruct structure-name
  (slot-name-1 default-value-1)
  (slot-name-2 default-value-2)
  ...
  (slot-name-n default-value-n))
```

- ✓ Example: define a starship structure, with slots for its name, speed, shield status, and number of photon torpedoes:

```
(defstruct starship
  (name nil)
  (speed 0)
  (shields 'down)
  (photon-torpedoes 10))
```

Defining structure types

- ✓ Defining a structure type automatically defines several other functions:

```
(defstruct starship
  (name nil)
  (speed 0)
  (shields 'down)
  (photon-torpedoes 10))
```

- x a constructor function `make-starship` which takes keyword arguments `:name`, `:speed`, `:shields`, `:photon-torpedoes` and returns an instance of the `starship` structure type
- x accessor functions `starship-name`, `starship-speed`, `starship-shields` and `starship-photon-torpedoes` which take a starship structure as argument and return the value of the corresponding slot
- x the accessor functions can also be used with `setf` to change the value of a structure slot
- x a type predicate `starship-p` to test if an object is a `starship` structure

Using structures in LISP

```
(defstruct starship
  (name nil)
  (speed 0)
  (shields 'down)
  (photon-torpedoes 10))
```

```
USER: (setq my-ship (make-starship :name 'enterprise))
#S(STARSHIP :NAME ENTERPRISE :SPEED 0 :SHIELDS DOWN
    :PHOTON-TORPEDOES 10)
```

```
USER: (starship-shields my-ship)
DOWN
```

```
USER: (setf (starship-shields my-ship) 'up)
UP
```

```
USER: (starship-shields my-ship)
UP
```

```
USER: (starship-p my-ship)
T
```

```
USER: (type-of my-ship)
STARSHIP
```

```
USER: (describe my-ship)
#<STARSHIP @ #xb4d5ae> is a structure of type STARSHIP.
It has these slots:
  NAME ENTERPRISE
  SPEED 0
  SHIELDS UP
  PHOTON-TORPEDOES 10
```

The type hierarchy in LISP

fig 12-1

Touretzky

Type predicates in LISP

- ✓ `typep` is a generic type predicate:

```
USER: (typep 3 'number)  
T
```

```
USER: (typep 3 'integer)  
T
```

```
USER: (typep 3 'float)  
NIL
```

```
USER: (typep 'foo 'symbol)  
T
```

- ✓ `type-of` is a function that returns a type identifier for a LISP object:

```
USER: (type-of 3)  
FIXNUM
```

```
USER: (type-of 'aardvark)  
SYMBOL
```

```
USER: (type-of '(bat breath))  
CONS
```

```
USER: (type-of "baloney")  
(SIMPLE-ARRAY CHARACTER (7))
```

I/O in LISP

- ✓ many AI applications interact with the user
- ✓ simple I/O functions in LISP are `read` and `print`
 - x `read` and `print` are the same as the LISP system functions used in the LISP interpreter's "read-eval-print" loop
 - `(print (eval (read)))` is what the LISP interpreter does!
 - x `read` takes no arguments and returns the next LISP expression typed in

```
USER: (read)
(here (is (a list)))
(HERE (IS (A LIST)))
```

- x `print` takes one argument and returns its value. As a side effect, the value is printed out, preceded by a carriage return and followed by a space

```
USER: (print '(here (is (a list))))
(HERE (IS (A LIST)))
(HERE (IS (A LIST)))
```

```
USER: (setf temp 100)
100
```

```
USER: (if (< -1 (print (- temp 98.6)) 1)
        'normal
        'abnormal)
```

```
1.4
ABNORMAL
```

I/O in LISP

- ✓ The format function permits fancy output formatting

```
(format t format-string arg1 arg2 ...)
```

- ✓ Ordinary characters in the format string are just printed out:

```
USER: (format t "Hello!")  
Hello!  
NIL
```

- ✓ Directive characters preceded by ~ in the format string have special effects

- x ~% in the format string causes a carriage return

```
USER: (format t "Hello! ~%I'm ready to start.")  
Hello!  
I'm ready to start.  
NIL
```

- x ~a in the format string causes the corresponding argument to be printed

```
(setf name 'kirsh symptoms '(fever rash nausea))
```

```
USER: (format t "Patient ~A has ~A symptoms: ~% ~A"  
            name (length symptoms) symptoms)
```

```
Patient KIRSH has 3 symptoms:  
(FEVER RASH NAUSEA)
```

- x there are many, many more format directives...