Lenguajes de Programación

- Pattern matching

Federico Olmedo
Ismael Figueroa

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Pattern matching

Pattern matching is a technique used to:

1. ***Check*** whether a given value ***matches a specified pattern.***
2. ***Deconstruct*** the matching value to access its data, by ***binding*** its “inner” elements to locally-scoped variables.
3. ***Perform*** an action using the recently-bound inner elements.

In general we use pattern matching when we know a given value falls into one of a finite number of cases—however the value is arbitrary and we must always handle it properly!

Pattern matching: syntax

SYNTAX: `(match target-expr [pattern expr …+] …)`

Similar to `cond`, a `match` expression has several clauses where `target-expr` is checked against. These clauses are checked in order, from top to bottom.

In a sense, pattern matching is a different kind of conditional expression, which is focused on the "shape" of the value we are inspecting.

Widely used in functional programming, pattern matching is available in more and more "real world" programming languages! In Python, the match statement is available since version 3.10

Pattern matching: basic example

```
(define (f v)
  (match v
    ["hola" (print "v es el string 'hola'")]
    [23     (print "v es el número 23")]))
```

Pattern matching: basic example

```
(define (f v)
 ①(match v
    ["hola" (print "v es el string 'hola'")]
    [23     (print "v es el número 23")]))
```

1 In Racket we use the **`match`** expression to perform pattern matching on a given value.

Pattern matching: basic example

```
(define (f v)
 ①(match v
   ②["hola"  (print "v es el string 'hola'")]
    [23      (print "v es el número 23")]))
```

1. In Racket we use the **`match`** expression to perform pattern matching on a given value.
2. We specify as many *clauses* as we need: the first element of a clause is the ***pattern***

Pattern matching: basic example

```
(define (f v)
 ❶(match v
    ["hola" (print "v es el string 'hola'")]
  ❷ [23    ❸(print "v es el número 23")]))
```

1. In Racket we use the **`match`** expression to perform pattern matching on a given value.
2. We specify as many *clauses* as we need: the first element of a clause is the ***pattern***
3. The second element of a clause is the ***action to perform* when the pattern is matched.**

Pattern matching: basic example

```
(define (f v)
  (match v
    ["hola" (print "v es el string 'hola'")]
    [23     (print "v es el número 23")]))
```

What are the results from executing:

```
(f “hola”)
```

- a `match` expression fails if no pattern is matched
- a `match` must be *exhaustive*: cover all possible cases

```
(f 23)
```

```
(f 1)
```

match: no matching clause for

Pattern matching: basic example

```
(define (f v)
  (match v
    ["hola" (print "v es el string 'hola'")]
    [23     (print "v es el número 23")]
    [else   (print "v es otra cosa")]))
```

What are the results from executing:

```
(f “hola”)

(f 23)

(f 1)
```

The _ pattern is used as an unnamed variable. It always matches. We will use `else` as with cond.

Pattern matching: the pattern language (simplified)

MATCHING LITERAL VALUES:

- Every literal value of a primitive datatype is a valid pattern (strings, numbers, etc).
- Any literal pair can be matched by using the **`cons`** constructor pattern, and using two sub-patterns for both the first and second element of the pair.
- Any literal list can be matched by using the **`list`** constructor pattern, and using as many sub-patterns for the fixed elements of the list.

```
(define (g v)
  (match v
    [(cons 1 (cons "hola" 'f)) (print "par literal")]
    [(list 1 "h" (cons 3 4))   (print "lista literal")]
    [else   (print "otro caso")]))
```

Pattern matching: the pattern language (simplified)

PATTERN VARIABLES:

Instead of trying to match literal values, *pattern variables* serve as wildcards to **match and bound** to whatever value appears in a specific position of a pattern.

- If the pattern is just a pattern variable, it will always match.

```
(define (h v)
  (match v
    [a (printf "anything: ~a" a)]))
```

- On pairs, pattern variables can be bound to the first or second element.

```
(define (j v)
  (match v
    [(cons a b) (printf "car: ~a cdr: ~a" a b)]
    [else (print "not a pair")]))
```

- On lists, we have several ways to match…

Pattern matching: working with lists

When working with lists we usually are interested in either:

- Matching the first element of the list, and ***the rest*** of the list. This is the classical pattern for recursive list-processing functions.
- Matching a fixed-length list, either with literal values or with pattern variables. This is how we will parse the source code for our interpreters.

Moreover, we can choose whether to treat the list as a nested pair, or as a list structure. We should choose whichever is more convenient for the task at hand.

Pattern matching: working with lists

- Matching the first element of the list, and ***the rest*** of the list. This is the classical pattern for recursive list-processing functions.

```
(define (my-length-cons l)
  (match l
    ['() 0]
    [(cons first rest) (+ 1 (my-length-cons rest))]))

(define (my-length-list l)
  (match l
    ['() 0]
    [(list first rest ...) (+ 1 (my-length-list rest))]))
```

Pattern matching: working with lists

- Matching a fixed-length list, either with literal values or with pattern variables. This is how we will parse the source code for our interpreters.

```
(define (match-binop l)
  (match l
    [(list '+ a b) (printf "sum of ~a and ~a" a b)]
    [(list '- a b) (printf "substraction of ~a and ~a" a b)]
    [(list '* a b) (printf "product of ~a and ~a" a b)]
```

Pattern matching: #:when clause

The optional **#:when** argument provides an additional condition that must be satisfied for the pattern to be matched. It has access to all bound variables in the pattern.

```
(define (sum-6 l)
  (match l
    [(list a b c) #:when (= 6 (+ a b c))
                  (print "sum is 6")]
    [(list a b c) (print "sum is not 6")]))
```

Pattern matching: predicate pattern

In any place we can use a pattern variable, we can instead apply a predicate **p** to that specific element using the `(? p)` pattern.

```
(define (starts-with-nested-list? l)
  (match l
    [(list) #f]
    [(cons first rest)
     (match first
       [(? list?) #t]
       [_ #f])]))

(define (is-second-odd? l)
  (match l
    [(list _ (? odd?)) #t]
    [else #f]))
```

Pattern matching: define by matching

Sometimes we know the expected pattern of a value, and wish to use pattern matching to deconstruct it and have access to its elements as named identifiers.

We can do this with the `match-define` **expression, which is also provided by the Play language,** simply as `def`.

```
(define (f l)
  (match l
    [(cons a b) (printf "first: ~a and rest: ~a" a b)]))
```

```
(define (f l)
  (match-define (cons a b) l)
  (printf "first: ~a and rest: ~a" a b))
```

```
(define (f l)
  (def (cons a b) l)
  (printf "first: ~a and rest: ~a" a b))
```

Exercises

Use pattern matching to define the following functions:

- `contains?`: returns whether a list contains a given value.
- `my-map`: applies a given function to each element of a list.
- `my-filter`: given a list, return a list with only the elements that satisfy a given predicate.
- `list-sum`: given a list of numbers, return the sum of all elements.
- `list-product`: given a list of numbers, return the product of all elements.
- count: given a list, return the number of elements that satisfy a given predicate.

Apply the function design methodology to all these exercises

Takeaways

- Pattern matching is a powerful technique to deconstruct and access data, in particular for processing compound data structures.

- Racket has a rich pattern language, which allows us to use literal values, pattern variables, and nested patterns amongst many other possibilities.

Lenguajes de Programación

- Inductive datatypes and recursion

Federico Olmedo
Ismael Figueroa

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

The natural numbers as an inductive set

$\mathbb{N}$ is the least set satisfying the following rules:

$$\frac{}{0 \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{n+1 \in \mathbb{N}}$$

Induction principle:

To prove a property P over the set of natural numbers we must:

- Prove that it holds for 0
- Prove that it holds for n+1 assuming it holds for n

$$\frac{P(0) \qquad \forall n.\ P(n) \implies P(n+1)}{\forall n.\ P(n)}$$

Recursion scheme:

To define a function f over the set of natural numbers we must:

- Define it for 0
- Define it for n+1 assuming we know its value for n

$$\frac{\begin{array}{lcl} f(0) & = & \ldots\ldots\ldots \\ f(n+1) & = & \ldots f(n) \ldots \end{array}}{\forall n.\ \text{"}f(n) \text{ is univocously defined"}}$$

Inductive sets

From the definition of an inductive set one can “blindly” derive its induction principle and recursion scheme.

Lists:

INDUCTIVE DEFINITION:

List is the least set satisfying the following rules:

$$\frac{}{\text{empty} \in \text{List}} \qquad \frac{l \in \text{List}}{\text{cons } v \; l \;\in\; \text{List}}$$

INDUCTION PRINCIPLE:

$$\frac{}{\forall l \in \text{List}.\; P(l)}$$

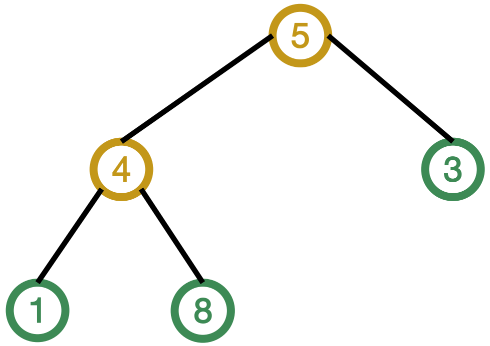
RECURSION SCHEME:

$$\begin{aligned} f(\text{empty}) &= \;\ldots\ldots\ldots \\ f(\text{cons } v \; l) &= \;\ldots f(l) \ldots \end{aligned}$$

$$\begin{aligned} \mathit{length}(\text{empty}) &= 0 \\ \mathit{length}(\text{cons } v \; l) &= 1 + \mathit{length}(l) \end{aligned}$$

Inductive sets

Binary trees:



INDUCTIVE DEFINITION:

BinTree is the least set satisfying the following rules:

$$\frac{}{(\mathsf{leaf}\ v)\ \in\ \mathsf{BinTree}} \qquad \frac{l, r\ \in\ \mathsf{BinTree}}{(\mathsf{in\text{-}node}\ v\ l\ r)\ \in\ \mathsf{BinTree}}$$

INDUCTION PRINCIPLE:

$$\frac{}{\forall bt \in \mathsf{BinTree}.\quad P(bt)}$$

RECURSION SCHEME:

$$f(\mathsf{leaf}\ v) \quad = \quad \ldots\ldots\ldots$$

$$\begin{aligned} height(\mathsf{leaf}\ v) &= \\ height(\mathsf{in\text{-}node}\ v\ l\ r) &= \end{aligned}$$

Inductive types and recursion in Racket

```
#lang play

#|
<BinTree> ::= (leaf <value>)
            |  (in-node <value> <BinTree> <BinTree>
|#
(deftype BinTree
  (leaf value)
  (in-node value left right))
```

1 The Play language provides **deftype** to create new inductive types with a given name.

Inductive types and recursion in Racket

```
#lang play

#|
<BinTree> ::= (leaf <value>)
            |   (in-node <value> <BinTree> <BinTree>
|#
(deftype BinTree
  (leaf value)
  (in-node value left right))
```

1 The Play language provides **deftype** to create new inductive types with a given name.

2 Each variant is defined by its own constructor function, which is always unique.

Inductive types and recursion in Racket

```
#lang play

#|
<BinTree> ::= (leaf <value>)
            |  (in-node <value> <BinTree> <BinTree>
|#
(deftype BinTree
  (leaf value)
  (in-node value left right))
```

1. The Play language provides **`deftype`** to create new inductive types with a given name.
2. Each variant is defined by its own constructor function, which is always unique.
3. The elements of each variant are defined as fields for each constructed variant type.

Each variant and its constructor can be used for pattern matching.
This is the standard way to work with inductive types and recursion.

Inductive types and recursion in Racket

In any data structure defined using `deftype` it holds that:

- All constructors are injective functions. For instance:

  `(equal? (leaf a) (leaf b))` is `#t` only when `(equal? a b)` is `#t`.

- Values built from different constructors are always different.

  For instance, for all a and b it follows that

  `(equal? (leaf a) (in-node b (…) (…)))` reduces to `#f`.

- The only way of building values of the data structure is via the provided constructors.

Inductive types and recursion in Racket

```
;; height :: BinTree -> number
;; Computes height of a binary tree.
(define (height bt)
  (match bt
    [(leaf _) 0]
    [(in-node _ left right)
     (+ 1 (max (height left) (height right)))]))
```

Each variant and its constructor can be used for pattern matching.
This is the standard way to work with inductive types and recursion.

Inductive types and recursion in Racket

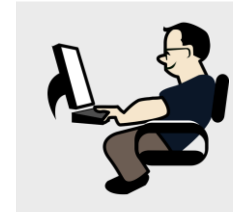
All recursive function over binary trees will have the following template, which follows from the **`deftype`**, which in turn follows from the grammar.

```
(define (func-to-define bt)
  (match bt
    [(leaf v) ...]
    [(in-node v left right) ...]))
```

```
(deftype BinTree
  (leaf value)
  (in-node value left right))
```

```
#|
<BinTree> ::= (leaf <value>)
           |  (in-node <value> <BinTree> <BinTree>
|#
```

Exercises



Function **sum-bin-tree**

Define function `(sum-bin-tree bt)`, which sums all elements from binary tree `bt`. Assume all values are numbers.

Function **max-bin-tree**

Define function `(max-bin-tree bt)`, which returns the element with maximum value in binary tree `bt`. Assume all values are numbers.

Capturing the recursive scheme

```
;; fold-bintree :: (Number -> A) (Number A A -> A) -> (BinTree -> A)
;; Fold over numeric binary trees.
(define (fold-bintree f g)
  (λ (bt)
    (match bt
      [(leaf v) (f v)]
      [(in-node v left right)
       (g v
          (fold-bintree f g left)
          (fold-bintree f g right))])))
```

Capturing the recursive scheme

```
;; fold-bintree :: (Number -> A) (Number A A -> A) -> (BinTree -> A)
;; Fold over numeric binary trees.
(define (fold-bintree f g)
  (λ (bt)
    (match bt
      [(leaf v) (f v)]
      [(in-node v left right)
       (g v
          (fold-bintree f g left)
          (fold-bintree f g right))]))))
```

1 Function to process recursive case for the first variant/constructor.

Capturing the recursive scheme

```
;; fold-bintree :: (Number -> A) (Number A A -> A) -> (BinTree -> A)
;; Fold over numeric binary trees.
(define (fold-bintree f g)
  (λ (bt)
    (match bt
      [(leaf v) (f v)]
      [(in-node v left right)
       (g v
          (fold-bintree f g left)
          (fold-bintree f g right))]))))
```

Recursive steps using the same folding function

1 Function to process recursive case for the first variant/constructor.

2 Function to process recursive case for the second variant/constructor.

Capturing the recursive scheme

3

```
;; fold-bintree :: (Number -> A) (Number A A -> A) -> (BinTree -> A)
;; Fold over numeric binary trees.
(define (fold-bintree f g)
 (λ (bt)
   (match bt
     [(leaf v) (f v)]
     [(in-node v left right)
      (g v
         (fold-bintree f g left)
         (fold-bintree f g right))])))
```

1 Function to process recursive case for the first variant/constructor.

2 Function to process recursive case for the second variant/constructor.

3 Function that recursively processes a binary tree, according to the specified behavior.

Using the recursive scheme

```
;; sum-bintree :: BinTree -> Number
;; Returns the sum of the elements of a numeric binary tree.
(define sum-bintree
  (fold-bintree identity +))

;; max-bintree :: BinTree -> Number
;; Returns the maximum value of a numeric binary tree.
(define max-bintree
  (fold-bintree identity max))
```

Exercises

Function **`contains-bintree?`**

Use **`fold-bintree`** to define function `(contains-bin-tree? bt v)`, which returns whether or not value v is found in the binary tree bt.

Follow the grammar

```
#|
<BinTree> ::= (leaf <value>)
            | (in-node <value> <BinTree> <BinTree>
|#
```

The grammar defines the inductive datatype.

Each variant requires a different constructor.

```
(deftype BinTree
  (leaf value)
  (in-node value left right))
```

This is mechanically translated into a `deftype`.

Each constructor has a folding function.

```
;; fold-bintree :: (Number -> A) (Number A A -> A) -> (BinTree -> A)
;; Fold over numeric binary trees.
(define (fold-bintree f g)
  (λ (bt)
    (match bt
      [(leaf v) (f v)]
      [(in-node v left right)
       (g v
          (fold-bintree f g left)
          (fold-bintree f g right))]))))
```

And the recursion scheme can also be mechanically derived.

Takeaways

Follow the Grammar!

When defining a function that operates on an inductive datatype, the definition should be patterned after the grammar of the datatype.

Following the grammar allows us to mechanically derive the `deftype`, the recursion scheme, and functions such as the parser for a given language.

Lecture Material

Bibliography

- PrePLAI: Introduction to functional programming in Racket [Sections 4-5]

- Essentials of Programming Languages (3rd Edition)
  Daniel P. Friedman
  [Chapter 1]

- Source code from the lecture [Download]