PROGETTAZIONE DI FUNZIONI RICORSIVE

→ Abbiamo visto che il Teorema di ricorsione consente di calcolare il GRAFICO di una funzione ricorsiva

→ Il grafico consente di verificare se la funzione è TOTALE (definita per tutti gli input, ossia Termina sempre)

→ In fase di progettazione di una funzione ricorsiva, per assicurarsi che sia Totale si può usare un altro metodo basato sul concetto di "RELAZIONE DI PRECEDENZA INDOTTA DALLA FUNZIONE"

Data una definizione ricorsiva di una funzione $f : A \to A$ la RELAZIONE di PRECEDENZA INDOTTA dalla definizione di f è la relazione $\prec_f$ tale che

$\forall x, y \in A . \; x \prec_f y$ se il calcolo di $f(y)$ richiama ricorsivamente $f(x)$

si dice

$x \prec_f y \Rightarrow x$ precede y

ESEMPIO

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-2) & \text{altrimenti} \end{cases} \qquad f : \mathbb{N} \to \mathbb{N}$$

abbiamo la seguente relazione di precedenza

$$x \prec_f y \equiv x = y - 2 \quad y > 1$$

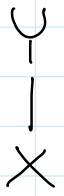
OSSIA x precede y SE $y > 1$ E $x = y - 2$

PERCHÉ CON $y > 1$ HO CHE CALCOLARE $f(y)$

RICHIEDE DI CALCOLARE RICORSIVAMENTE $f(y-2)$
x

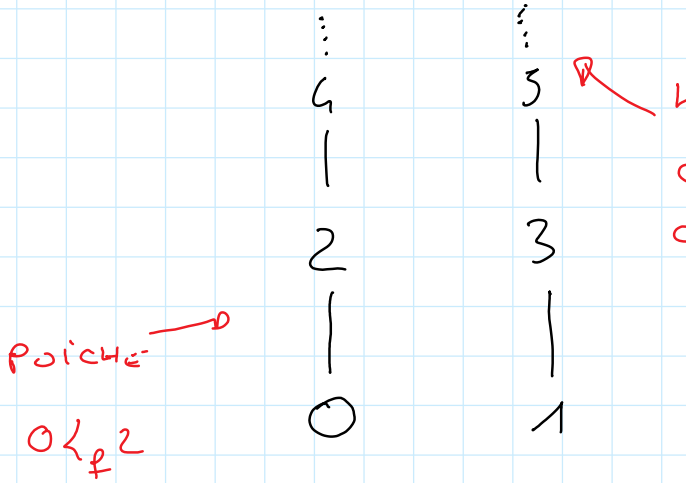
Disegnamo questa relazione:

per ogni coppia di valori x, y Tali che $x \lhd_f y$ disegno



DISEGNO QUELLO CHE PRECEDE SOTTO L'ALTRO

LO FACCIO PER TUTTI I VALORI DI m



HO OTTENUTO DUE CATENE DI VALORI CHE CRESCONO VERSO L'ALTO ALL'INFINITO

POICHÉ $0 \lhd_f 2$

QUESTO DIAGRAMMA RAPPRESENTA TUTTE LE POSSIBILI CHIAMATE RICORSIVE

→ IL FATTO CHE NON CI SIANO SEQUENZE DISCENDENTI INFINITE ASSICURA CHE TUTTE LE CHIAMATE ALLA

FUNZIONE (PER QUALSIASI INPUT)
TERMINO SEMPRE (PRIMA O
POI RAGGIUNGONO UN CASO BASE)

ESEMPIO

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 + f(n \div 2) & n \neq 1 \end{cases} \qquad f : \mathbb{N} \to \mathbb{N}$$

DISEGNAMO $\mathcal{G}_f$

0 — 1 — 1

2 — 1 — 3 — 1 — 4 — 1 — 5 — 1 — ⋮

$\mathcal{G}_f$ = IN QUESTO CASO HO UNA SEQUENZA DISCENDENTE INFINITA IN $\mathcal{G}_f$

⇓

f NON È TOTALE (IN CERTI CASI NON TERMINA)

ESEMPIO (successione di Fibonacci)

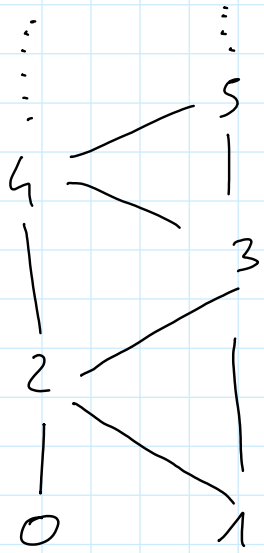
0 1 1 2 3 5 8 .....

0 1 2 3 4 5 6 ....

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

DISEGNAMO $\{_{fib}$

OGNI $n > 1$ HA 2 PRECEDENTI $n-1$ $n-2$



2 È PRECEDUTO SIA DA 1 CHE DA 2 PERCHÉ HA 2 CHIAMATE RICORSIVE

ANCHE IN QUESTO CASO NON HO SEQUENZE DISCENDENTI INFINITE!!

TOTALE!!

martedì 21 novembre 2017 11:45

PROGRAMMAZIONE RICORSIVA E FUNZIONALE IN CAML LIGHT!

→ VEDERE LA DISPENSA DISPONIBILE SULLA PAGINA WEB DEL CORSO

→ SCARICARE L'INTERPRETE CAML-LIGHT (IL LINK È SULLA PAGINA WEB DEL CORSO, FATE ATTENZIONE A SCARICARE CAML-LIGHT E NON OCAML)

→ L'interprete CAML valuta espressioni fornendone il risultato

→ il prompt # si aspetta l'espressione

→ l'espressione deve finire con ;;

```
# 10+12;;
- : int = 22
```

NON VA SCRITTO... E' GIÀ PRESENTE

← RISPOSTA DATA DALL'INTERPRETE

→ l'interprete risponde con 3 informazioni

- → NOME DEFINITO (- per nessun nome)
- → TIPO DELL'ESPRESSIONE (int, nell'esempio)
- → VALORE (risultato calcolato)

```
# 33 ;;
- : int = 33

# 10+12 ;;
- : int = 22

# 3.4 ;;
- : float = 3.4
```

INFERENZA DI TIPI

Caml INFERISCE (ossia deduce) il Tipo dell'espressione dai valori e dalle operazioni in essa contenute

3+2 ha Tipo int perché è una somma di numeri interi

IL LINGUAGGIO È FORTEMENTE TIPATO

↳ ogni espressione ha un Tipo ben definito e le operazioni devono essere fatte rispettando i Tipi degli operandi.

LE OPERAZIONI ARITMETICHE ESISTONO IN DUE VERSIONI DIVERSE: TRA INTERI (int) E TRA FRAZIONARI (float)

su interi: +, -, *, /, mod ← modulo

su float: +., -., *., /.

CI VUOLE IL PUNTO!!

```
# 3.4 +. 6.2;;
- : float = 9.6

# 3.4 + 6.2;;
```

ERRORE DI TIPO!!

Si possono scrivere anche espressioni logiche (esiste il Tipo bool per memorizzare valori di verità)

- true e false sono valori
- operazioni di confronto

  <, >, =, !=, >=, ...

  - ATTENZIONE = E NON ||
    == COME IN C
- OPERATORI LOGICI

  &&, ||, not

  - ATTENZIONE, not E NON ! COME IN C

ESempi

```
# True ;;
- : bool = True

# 3 < 5 && 1 = 2 ;;
- : bool = false

# not True ;;
- : bool = false
```

Possiamo associare un'espressione a un NOME

Tramite l'operatore LET

ATTENZIONE: NOME, NON VARIABILE!!

```
# let x = 10+2;;
x : int = 12
```

IN CAML NON ABBIAMO UNO STATO COME IN C

→ ABBIAMO SOLO UN AMBIENTE COSTITUITO DA ASSOCIAZIONI NOME - VALORE

int z 1074
int y 141
int x 12

AMBIENTE

in realtà CAML si ricorda anche i Tipi dei nomi

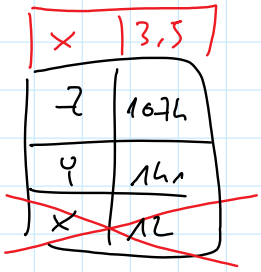
I NOMI NON SONO VARIABILI.

NON SI PUÒ ASSEGNARE UN NUOVO VALORE A UN NOME GIÀ DEFINITO.

POSSO PERÒ RIDEFINIRE UN NOME GIÀ DEFINITO FACENDO DI NUOVO LET

```
# let x = 3.5;;
x : float = 3.5
```

CAML ELIMINA LA VECCHIA ASSOCIAZIONE E LA SOSTITUISCE CON LA NUOVA



1) x PUÒ ESSERE RIDEFINITO CON UN TIPO DIVERSO!

2) NON POTRÒ MODIFICARE IL VALORE DI x ALL'INTERNO DI UN'ESPRESSIONE (NON POSSO, AD ESEMIO, SCRIVERE UN CICLO)

DOPO AVER DEFINITO UN NOME POSSO USARLO IN UNA SUCCESSIVA ESPRESSIONE

```
# let x = 2+5;;
x : int = 7
# let y = x + 8;;
y : int = 15
```

martedì 21 novembre 2017 12:32

DEFINIRE FUNZIONI

L'operatore LET può essere usato per definire funzioni.

↳ LA PARTICOLARITÀ DEI LINGUAGGI FUNZIONALI (COME CAML) È CHE LE FUNZIONI SONO ANCH'ESSE VALORI!

```
# let f x = x+1;
f : int -> int = <fun>
```

- `f` — NOME
- `int -> int` — TIPO DELLA FUNZIONE (INFERITO)
- `<fun>` — VALORE FUNZIONE

DEFINISCE LA FUNZIONE f CON PARAMETRO X CHE RESTITUISCE X+1

COME HA FATTO A INFERIRE

int → int

1) `let f x = ....`

↳ SUGGERISCE CHE f È FUNZIONE

..... → .....

2) SICCOME NEL CORPO HO X+1

DEDUCO CHE X È DI TIPO INT

= SOMMA TRA INTERI

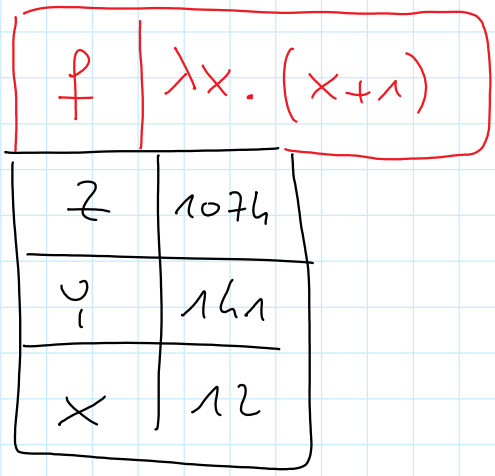
int → ....

3) SICCOME X+1 È UN'ESPRESSIONE DI TIPO int POSSO DEDURRE

int → int

DEFINENDO UNA FUNZIONE SI CREA UN'ASSOCIAZIONE NELL'AMBIENTE

```
# let f x = x+1 ;;
```



AMBIENTE

λ LAMBDA

λx.(x+1)

RAPPRESENTA LA FUNZIONE CHE DATO X CALCOLA X+1

L'APPLICAZIONE DELLA FUNZIONE CONSISTE NEL PRENDERE L'ARGOMENTO, SOSTITUIRLO AL PARAMETRO E CALCOLARE IL RISULTATO DELL'ESPRESSIONE

```
# f 2 ;;
- : int = 3
```

CORRISPONDE A FARE

2

λx.(x+1) = 2+1 = 3

$\lambda x \cdot (x + 1) \quad = \quad 2 + 1 = 3$

ATTENZIONE

La dichiarazione di funzione

```
# let f x = x+1 ;;
f : int -> int = <fun>
```

ha come effetto che viene aggiunta nell'ambiente una associazione per f (come si capisce dalla risposta dell'interprete)

Non viene aggiunta nell'ambiente NESSUNA ASSOCIAZIONE per il parametro x !!

→ x È USATO LOCALMENTE ALLA FUNZIONE

→ SE x ERA GIÀ STATO DEFINITO NELL'AMBIENTE, LA x LOCALE ALLA FUNZIONE NON INTERFERISCE (LA x DEFINITA PRIMA RIMANE DISPONIBILE IMMUTATA)

FUNZIONI CHE PREVEDONO PIU' DI UN PARAMETRO.

ES. SOMMA DI 2 NUMERI

1° MODO) USANDO L'ESPRESSIONE COPPIA!

SI DEFINISCONO USANDO L'OPERATORE

, (VIRGOLA)

COPPIE E ENNUPLE

L'operatore , consente di costruire coppie di espressioni

```
# 3 , 5 ;;
- : int * int = 3, 5
```

= PRODOTTO CARTESIANO

UNA COPPIA HA COME TIPO IL PRODOTTO CARTESIANO * DEI TIPI DEI SINGOLI ELEMENTI.

↳ POSSO USARE fst E snd PER OTTENERE IL 1° E IL 2° ELEMENTO DI UNA DATA COPPIA

DI UNA DATA COPPIA

- POSSO DEFINIRE ENNUPLE USANDO LA , PIU' VOLTE

ESEMPI:

```
# 6 , 4.2 ;;
- : int * float = 6 , 4.2

# 3+2 , 4-1 ;;
- : int * int = 5 , 3

# fst (3,1) ;;
- : int = 3

# snd (3,1) ;;
- : int = 1

# p = (6.2 , 5);;
p : float * int = (6.2 , 5)

# fst p ;;
- : float = 6.2

# 3+2 , 6.4+.1.0 , 9 , 8.4 ;;
- : int * float * int * float = 5 , 7.4 , 9 , 8.4
```

CI VOGLIONO LE PARENTESI PERCHE' L'APPLICAZIONE DELLE FUNZIONI (fst e snd) HA LA PRECEDENZA SULLA ,

ESEMPIO di FUNZIONE CON DUE PARAMETRI come COPPIA

```
# let somma (x,y) = x+y ;;
somma : int * int -> int = <fun>

# somma (3,2);;
- : int = 5
```

LA FUNZIONE IN REALTÀ PREVEDE UN UNICO PARAMETRO DI TIPO int * int (UNA COPPIA)