Soluzione
Prova in itinere 27/04/2016

Luca Parolari*
II anno - informatica

April 18, 2018

Esercizio 1. Con la notazione $C^L_{L_1,L_2}$ indichiamo un compilatore da L_1 a L_2 scritto in L. Con $I^L_{L_1}$ indichiamo un interprete scritto in L per il linguaggio L_1.

Supponiamo che A sia il linguaggio della macchina M e supponiamo che X sia un nuovo linguaggio che desideriamo implementare su M. Possiamo fare questo:

a) scegliendo un opportuno $S \subseteq X$;

b) creando un compilatore da S in A scritto in A;

c) ... e che altro?

Descrivere formalmente l'intero procedimento.

Avendo implementato X su M, come potremmo ottenere un'implementazione di X per una macchina M_0 il cui linguaggio è A_0?

Soluzione Come si fa???[1]

Esercizio 2. Nello scope delle dichiarazioni:

```
int n;
string s;
int g(int x, real y) {...}
```

si consideri l'espressione $g(f(n), f(s))$. Si diano ipotesi sul linguaggio e/o sul nome f affinché tale espressione sia corretta rispetto ai tipi.

Soluzione Vedendo questo codice, si suppone che il linguaggio sia sicuramente ad alto livello, ma oltre a ciò, si suppone anche che sia abbastanza evoluto da supportare l'overloading di funzioni.

Un possibili prototipo della funzione f può essere il seguente:

*luca.parolari@studenti.unipr.it

[1] Non ho capito il testo dell'esercizio e nemmeno come si potrebbe risolvere. Chi riuscisse a farlo, potrebbe condividere la soluzione sulla mailing list?

```
int f(int);
real f(string s);
```

che sfrutta l'overloading per la risoluzione delle due chiamate.

Inoltre, si suppone che il passaggio dei parametri avvenga per valore ed anche il modo di ritorno sia value return.

Esercizio 3. Si consideri il seguente programma:

```
void swap (int a, int b) {
   int temp;
   temp = a;
   a = b;
   b = temp;
}
void main () {
   int value = 2;
   int list[5] = {1, 3, 5, 7, 9};
   swap(value, list[0]);
   swap(list[0], list[1]);
   swap(value, list[value]);
}
```

Per ognuna delle seguenti modalità di passaggio dei parametri, si dica quali sono i valori delle variabili value e list dopo ognuna delle tre chiamate a *swap*: per valore, per riferimento, per nome, per valore-risultato.

Soluzione Da fare...[2]

Esercizio 4. Si assuma che in un generico linguaggio imperativo a blocchi, il blocco A contenga una chiamata della funzione f . Il numero dei record di attivazione (RdA) presenti a run-time sulla pila fra il RdA di A e quello della chiamata di f è fissato staticamente (ovvero è determinable a tempo di compilazione) o invece può variare dinamicamente (e dunque può essere determinato solo a tempo di esecuzione)? Motivare la risposta.

Soluzione In generale il numero di record di attivazione tra A e quello della chiamata a f non può essere noto a priori in un linguaggio che permette la ricorsione. f infatti potrebbe essere proprio una funzione ricorsiva che richima se stessa un numero n di volte non noto a tempo di compilazione.

Se invece il linguaggio non permettesse la ricorsione il compilatore sarebbe in grado di calcolare il numero massimo di RdA presenti sullo stack, in quanto ogni possibile chiamta da f a altre funzioni sarebbe determinabile a tempo di compilazione.

Nota: anche con un ciclo che richiama più volte una funzione g, un solo RdA è inserito nello stack, che, terminata la funzione, viene distrutto.

Esercizio 5. Si consideri la seguente definizione di tipo record:

[2]Per questioni di tempo questa soluzione non è pubblicata assieme al resto delle soluzioni. *Potrebbe* essere svolta e pubblicata più avanti.

```
type S = struct {
   int x;
   char y;
};
```

Si supponga che un int sia memorizzato su 2 byte, un char su 1 byte, su un'architettura a 16 bit con allineamento alla parola. In un blocco viene dichiarato un vettore:

```
S A [10];
```

Indicando con **PRDA** il puntatore all'RdA di tale blocco, e con **ofst** l'offset tra il valore di PRDA e l'indirizzo iniziale di memorizzazione di A, si dia l'espressione per il calcolo dell'indirizzo dell'elemento $A[i].y$.

Soluzione Definiamo alcune funzioni ausiliarie per la risoluzione del problema:

$s\_no\_align(S) = s(int) + s(char)$, calcola la dimensione della struct S senza lo spazio di pad.
$align(S) = s\_no\_align(S)\ mod\ s(word)$, calcola lo spazio di pad della struct S.
$s(S) = s(int) + s(char) + align(S)$, calcola la dimensione complessiva della struct S.

$$ind(A[i].y) = PRDA + ofst + i * s(S) + s(int)$$

Esercizio 6. Sono date le seguenti definizioni di funzione:

```
int fact_falso(int n) {
   if (n == 0) return 1;
   else return fact_falso(n + 1);
}
int fact_vero(int n){
   if (n == 0) return 1;
   else return n * fact_vero(n - 1);
}
```

Una certa implementazione del linguaggio si comporta nel modo seguente. Alla chiamata `fact_falso(1)`, non risponde, rimanendo in un loop infinito. Alla chiamata `fact_vero(-1)` risponde dopo qualche tempo con Stack overflow during evaluation, abortendo l'esecuzione. Si dia una spiegazione motivata di questi due fatti.

Soluzione La prima versione del fattoriale rimane in un loop infinito senza terminare mai in quanto viene sfruttata la tecnica denominata **tail recursion**.

Per avere la *tail recursion* l'ultima operazione effettuata nella funzione ricorsiva deve essere proprio la chiamata ricorsiva: in questo modo il calcolatore ha già eseguito tutti i calcoli che servono per la costruzione del risultato e non ha bisogno di tenere in memoria l'*RdA* di ogni chiamata. Riciclando sempre lo stesso RdA `fact_falso` entra in un ciclo infinito e continua a chiamare se stesso con $n + 1$, senza dare Stack Overflow.

La seconda versione del fattoriale invece non sfrutta la tail recursion, nel `return` infatti vi è una moltiplicazione: il calcolatore ha bisogno di memorizzare il valore di n sullo stack prima di effettuare la chiamata ricorsiva, di modo che, quando la funzione arriva al caso base

ed inizia a "sgonfiare" lo stack delle chiamate, ad ogni passo vi sia il valore memorizzato di n da moltiplicare con quello appena restituito dalla chiamata. E' evidente che ciò, dopo un certo tempo di esecuzione, generi Stack Overflow, in quanto ad ogni chiamata un RdA è messo sullo stack.

Esercizio 7. Si spieghi la differenza tra iterazione determinata ed iterazione indeterminata. Si dica, motivando la risposta, se un linguaggio con allocazione statica della memoria può contenere un comando di iterazione indeterminata.

Soluzione In un linguaggio di programmazione con iterazione determinata è garantito che un certo frammento di codice che viene iterato termini. In particolare, è garantito che il numero di iterazioni siano esattamente

$$ic = \lfloor \frac{fine - inizio + step}{step} \rfloor$$

ic o *iteration count* è il contatore di iterazioni, determinato a tempo di esecuzione. Ciò è determinato non appena un costrutto di iterazione determinata è eseguito: le espressioni *inizio* e *fine* e *step* vengono valutate e assegnate alle relative variabili rese indisponibili di modifica al programmatore. Il "contatore di iterazone" i viene anch'esso valutato e poi incrementato dal linguaggio di *step*: il programmatore può accedere solo in lettura. La semantica del costrutto di iterazione determinata quindi è *ripeti ic volte il corpo del for*. In linea di principio queso costrutto viene eseguito da $i = 1$ fino a che i è strettamente maggiore di $fine$, ma ovviamente ogni linguaggio è libero di prendere le scelte progettuali che ritiene adeguate.

La differenza fondamentale con il costrutto di iterazione indeterminata è che questo è appunto indeterminato: il linguaggio non si prende in carico nessun tipo controllo. Un costrutto di iterazione indeterminata può essere quindi visto come un while strutturato, dove una possibile sintassi può essere:

```
for(expr1; expr2; expr3)
   body
```

e la relativa semantica:

1) Valuta *expr*1;
2) Valuta *expr*2 (deve essere di tipo booleano);
3) Se *expr*2 è vera esegui *body*;
4) Valuta *expr*3;
5) Riparti da 2)

Esercizio 8. In un linguaggio di programmazione non meglio specificato, l'espressione $1 + 3.14$ valuta all'intero 4, mentre l'espressione 3.14+1 valuta al numero razionale 4.14000034332275390625. Si espongano deduzioni plausibili a partire da queste ipotesi.

Sotto le stesse ipotesi, si dica a cosa valutano le seguenti quattro espressioni: $1 + 1$, $1 + 0.75$, $0.75 + 1$, $0.75 + 0.75$.

Soluzione Possiamo supporre che in questo linguaggio sia definito l'operatore binario infisso + in questo modo:

1) $+ : int \times int \to int$

2) $+ : float \times float \to float$

Avendo definito + come una funzione in overloading, la scelta della funzione utilizzabile alla chiamata è effettuata in base al tipo dell'operando sinistro, mentre il resto è convertito implicitamente di conseguenza.

Inoltre, si suppone che la rappresentazione di un float sia IEEE754 a precisione singola, con 24 bit di mantissa in cui vi sono $log_{10}(2^{24}) \approx 7$ cifre decimali significative.

Dalle ipotesi precedenti segue che

1) $1 + 1 = 2$

2) $1 + 0.75 = 2$

3) $0.75 + 1 = 1.75$

4) $0.75 + 0.75 = 1.5$

notando che 3) e 4) sono valutate esattamente (senza cifre decimali "strane"), perché, a differenza di 4.14, 0.75, 1.5 e 1.75 non sono periodici in base 2, e possono essere rappresentati esattamente con un float. Infatti, ad esempio, $1.75 = 1 + 1/2 + 1/4$, cioè 1.11 in base 2.