Sviluppare Programmi Corretti I: Programmi Iterativi

Metodologie di Programmazione
Ivano Salvo
Sapienza Università di Roma
email: salvo@di.uniroma1.it

Anno Accademico 2013-14

Presentazione

La presente dispensa si propone di presentare lo sviluppo di programmi iterativi corretti, utilizzando *asserzioni logiche*.

Verranno presentate metodologie per *specificare* il comportamento dei programmi, e per sviluppare programmi che *realizzano* tali specifiche.

Vengono presentati un buon numero di esercizi, il cui svolgimento corretto dovrebbe assicurare un comodo passaggio dell'esame. Esercizi particolarmente difficili vengono segnalati con il simbolo ★, mentre gli esercizi contrassegnati dal simbolo ♣ sono in effetti "complementi" alla teoria presentata e quindi sono utili soprattutto ad una assimilazione "attiva" dei concetti. Il simbolo ★ può anche comparire a segnalare una sezione che richiede un certo sforzo e che, tipicamente, non è stata presentata nelle lezioni in aula, non è programma d'esame ed quindi riservata a proporre ulteriori stimoli agli studenti interessati e curiosi.

Si tratta di una bozza e quindi è probabile la presenza di imperfezioni o errori. I lettori sono invitati a segnalare le imperfezioni e gli errori rilevati al docente, in modo da migliorare la presente dispensa.

1 Astrazione, Problemi, Esecutori

In questa prima sezione sarà introdotto l'utilizzo di asserzioni logiche per la progettazione e la verifica di correttezza dei programmi.

Ogni componente di programma è il risultato di un processo che comincia con l'*individuazione* di un'*astrazione*, prosegue con la *descrizione* delle sue caratteristiche, cioè la *specifica*, si concretizza con l'*implementazione* che *realizza* la specifica e termina con la *verifica* che l'implementazione effettivamente soddisfi la specifica, via dimostrazioni formali o più comunemente attraverso il testing.

Ogni linguaggio di programmazione offre al programmatore una *macchina virtuale* (o *esecutore*) che ha alcune capacità elementari, nei cui termini i programmi dovranno descrivere le soluzioni dei problemi. Inoltre, ogni linguaggio offre al programmatore dei meccanismi per estendere le capacità della macchina virtuale, attraverso la definizione di nuove capacità (ossia, nuove astrazioni).

In questa prima parte ci concentreremo sulle *funzioni* (o se preferite essere un po' pomposi, sulle *astrazioni procedurali*), che in Java sono "tecnicamente" *metodi statici*.

1.1 La Macchina Virtuale TinyJava

Per rimarcare il concetto di esecutore, inizieremo il nostro viaggio, supponendo che l'esecutore a cui facciamo riferimento nella soluzione dei problemi, sia un esecutore molto semplice, in grado solo di eseguire iterazioni (con l'unico ciclo `while`), scelte (costrutto condizionale `if`), di sommare 1 a una variabile, e di confrontare se il valore di due espressioni sia uguale o diverso. Si tratta di un sottoinsieme di Java, che chiameremo amichevolmente TinyJava, che quindi, più precisamente comprende:

variabili un insieme di variabili *Var*, che vengono dichiarate all'inizio di ogni blocco di programma. La dichiarazione associa a una variabile un *tipo*. Considereremo solo `int` e `boolean` come possibili tipi;

espressioni aritmetiche indicheremo con *ExpA* l'insieme delle delle *espressioni aritmentiche* che si possono costruire a partire da costanti numeriche, variabili intere, usando unicamente l'operatore successore `+1`;

espressioni booleane indicheremo con *ExpB* l'insieme delle delle *espressioni booleane* che si possono costruire a partire da costanti booleane (`true` e `false`), variabili booleane e le espressioni atomiche che testano l'uguaglianza di due espressioni e_1`==`e_2 e e_1`!=`e_2, dove $e_1, e_2 \in Exp$, l'insieme di tutte le espressioni ($Exp = ExpA \cup ExpB$);

assegnazione il comando di assegnazione v`=`e, con $v \in Var$ e $e \in Exp$ che permette di modificare lo stato della memoria, modificando il contenuto della cella di memoria riferita dalla variabile v con il valore calcolato per l'espressione e;

scelta il comando condizionale `if (`e`)` C_1 `else` C_2 che permette di eseguire il comando C_1 se $e \in expB$ valuta a `true`, e C_2 altrimenti;

iterazione il comando iterativo `while (`e`)` C che ripete l'esecuzione del comando C fino a quando la condizione booleana e valuta a `true`.

sequenza dati n comandi $C_1, C_2, \ldots, C_n$, possiamo considerare il comando sequenza $C = \{C_1; C_2; \ldots; C_n\}$ che esegue prima C_1, poi C_2, ..., fino a C_n.

1.2 Cominciando con TinyJava: Somma di due Numeri

Cominciamo con il risolvere un semplicissimo problema, che dovrebbe allietarvi riconducendovi a quando, nell'infanzia, avete cominciato a contare sulle dita delle mani.

Problema 1: *Scrivere una funzione* TinyJava *che calcola la somma dei due numeri naturali.*

Immaginiamo inizialmente di essere noi stessi gli esecutori dell'algoritmo e di avere a disposizione carta, matita e gomma. L'idea è molto semplice: è sufficiente sommare 1 al primo numero un numero di volte pari al secondo al numero. Potremmo procedere nel seguente modo:

1. scriviamo su un foglio di carta i due numeri con la matita. Appare subito evidente che conviene dare loro un nome, per indicarli con facilità nella descrizione della procedura, diciamo n ed m;
2. cancelliamo con la gomma (la casella contenente) m e riscriviamo al suo posto (il valore) $m + 1$;
3. cancelliamo (la casella chiamata) n e riscriviamo in essa (il valore) $n - 1$;
4. se n è zero, significa che abbiamo sommato 1 a m già n volte, e abbiamo quindi finito, e la somma ora risulta scritta nella casella dedicata ad m. Viceversa torniamo al passo 2 e continuiamo le operazioni.

La soluzione sopra descritta ha (almeno) un paio di problemi:

- non funziona nel caso in cui n sia 0 all'inizio (verificate);
- il nostro esecutore protesterà, perché la procedura fa riferimento a un'azione che non sa compiere: il predecessore.

Il primo problema si aggiusta, osservando che è necessario spostare il controllo sul valore di n prima delle operazioni di cancellazione e riscrittura fatte ai passi 2 e 3.

Per il secondo problema, possiamo momentaneamente supporre di avere a disposizione un esecutore capace di fare il predecessore, riservandoci in un secondo momento di definire una funzione `pred` che permetta al nostro algoritmo di funzionare correttamente. Questo modo di procedere è assolutamente *consigliato* e permette di riferirsi all'occorrenza a un esecutore

L'algoritmo sopra descritto può essere facilmente codificato nel nostro TinyJava, come in Fig. 1.

1.3 Asserzioni Logiche

Osservate che abbiamo corredato il programma con commenti (tutto il testo tra i simboli `/*` e `*/`) che esprimono *asserzioni logiche* sui valori delle variabili del programma nel punto in cui l'asserzione logica è scritta.

```
public static int somma(int m, int n){
/* REQUIRE: m,n>=0.
 * EFFECTS: returns m+n.
 */
  while (n!=0) {
      /* INV: m0+n0=m+n. T=n. */
   m=m+1;
   n=pred(n);
  }
  /* AF: m=m0+n0 */
  return m;
}
```

Figura 1: Funzione per calcolare la somma di due numeri naturali

Noi useremo una "pseudo-logica" che userà con un certo "informalismo" un vernacolo matematico che usa formule atomiche (con operatori relazionali come $=, \leq, <$, etc., operatori aritmetici come $+, \times$, etc., gli usuali connettivi logici $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implica) e i quantificatori $\forall$ (per ogni) e $\exists$ (esiste)).

Nel testo (in particolare nelle dimostrazioni) indicheremo col carattere v il valore di una variabile `v`. Ci risulterà comodo usare la notazione v_0 per indicare il valore della variabile `v` prima dell'inizio di un comando (usualmente iterativo), e con v' il valore di una variabile dopo l'esecuzione di un comando.

A volte, useremo lettere greche $\varphi, \psi, \phi, \ldots$ per indicare un'asserzione logica e la notazione $\varphi[v]$ per mettere in evidenza che φ dipende dalla variabile v. Talvolta, useremo la notazione $\varphi[e/v]$ per indicare la formula logica ottenuta rimpiazzando le eventuali occorrenze di v in φ con l'espressione e.

Con abuso di linguaggio, se b è una espressione booleana, indicheremo con b anche l'asserzione logica definita dall'espressione booleana b.

1.3.1 Precondizioni

La clausola REQUIRE esprime le *precondizioni* sui valori dei parametri di ingresso. Nella funzione `somma`, essa semplicemente esprime che ogni valore positivo va bene, e questo è coerente col fatto che la somma è una funzione totale sui numeri naturali. Osserviamo che una funzione deve garantire risultati corretti *solo quando* i parametri di ingresso soddisifino le precondizioni. Vedremo più avanti un modo elegante (le cosidette *eccezioni*) di trattare il caso in cui i parametri non soddisfino le precondizioni.

1.3.2 Postcondizioni

La clausola EFFECTS esprime le *postcondizioni* sui valori ritornati dalla funzione, ed eventualmente anche su altri effetti (una funzione, infatti, potrebbe modificare anche lo stato della memoria del programma chiamante, anche se si tratta di una

pratica di programmazione da evitare). Precondizioni e postcondizioni in qualche modo sono una sorta di contratto tra la funzione (o se preferite il progettista della funzione) e il chiamante (o se preferite il programmatore utilizzatore).

1.3.3 Invarianti

Ogni ciclo dovrebbe contenere una clausola INV, che esprime l'*invariante del ciclo*, ossia una condizione logica soddisfatta in tutte le iterazioni del ciclo. Intuitivamente, un invariante cattura la regolarità che è connaturata alla possibilità di ripetere alcune istruzioni nell'esecuzione di un calcolo.

Nel nostro caso, l'invariante $m_0 + n_0 = m + n$ esprime che durante l'esecuzione del ciclo la somma dei valori contenuti nelle variabili `m` ed `n` rimane costante, ed uguale al valore da calcolare. Osserviamo che ci sono numerose proprietà invarianti. Ad esempio, nell'iterazione della funzione `somma`, anche $m \geq 0$ è una proprietà invariante. Infatti $m \geq 0$ è implicato dalle precondizioni e, durante il ciclo, m viene solo incrementato. Anche se si tratta di una proprietà interessante, tuttavia, non ci aiuta a valutare la correttezza della funzione `somma`.

Viceversa, dalla proprietà $m_0 + n_0 = m + n$ e dalla negazione della guardia del ciclo, che implica $n = 0$, otteniamo, per sostituzione, l'asserzione finale $m = m_0 + n_0$, che ci autorizza a ritornare il valore di `m`, sicuri che esso corrisponda alla somma da calcolare. Le idee sopra esposte hanno portata generale e si concretizzano nei seguenti due teoremi.

Teorema 1.1 *Indichiamo con v i valori dell'insieme delle variabili considerate in un programma. Una formula $\varphi[v]$ è invariante per il ciclo* `while (`b`)` C *se:*

1. *$\varphi[v]$ è soddisfatta prima dell'esecuzione del ciclo;*
2. *Assumendo $\varphi[v] \wedge b[v]$ prima dell'esecuzione del comando C, ho che l'asserzione $\varphi[v']$ è soddisfatta dopo l'esecuzione di C.*

Nell'esempio della funzione `somma`, è sufficiente far vedere che chiaramente $m_0 + n_0 = m + n$ all'ingresso del ciclo in quanto, banalmente, in quel punto $m_0 = m$ e $n_0 = n$. Inoltre, $m' + n' = (m + 1) + (n - 1) = m + n$. Questo, ovviamente, a patto che si possa fare il predecessore. Qui ci aiuta la guardia $n \neq 0$: infatti quando $n = 0$, il valore della funzione predecessore sui naturali non è definito.

Teorema 1.2 *Se φ è un invariante per* `while (`b`)` C *e $\varphi \wedge \neg b \Rightarrow \psi$, allora l'asserzione ψ è soddisfatta dopo la terminazione del ciclo.*

Nel nostro esempio, l'*asserzione finale* che assicura la correttezza della funzione è $\psi[m] \equiv m = m_0 + n_0$. Chiaramente l'invariante $m + n = m_0 + n_0$, insieme alla negazione della guardia $n = 0$, per sostituzione (mettendo 0 al posto di n) implicano l'asserzione finale.

1.3.4 Funzione di Terminazione

Il Teorema 1.2 contiene un'ipotesi di cui non abbiamo ancora parlato, e cioè la *terminazione* di un costrutto iterativo. Credo tutti voi abbiate già fatto l'esperienza di scrivere un programma che non termina. Un simpatico esempio di iterazione che non termina è la seguente funzione `beato`:

```
public static void beato(){
    while (true){
    }
}
```

che è il programma che non fa nulla per l'eternità.

Per dimostrare formalmente che una iterazione termina, useremo lo strumento concettuale della *funzione di terminazione*, che usualmente indicheremo con τ. Una funzione di terminazione per un ciclo è una funzione che associa ai valori delle variabili di un programma v un numero naturale ed è strettamente decrescente ad ogni iterazione. Siccome nei numeri naturali non esistono successioni infinite decrescenti (un principio connesso al principio di induzione), formalmente, abbiamo il seguente:

Teorema 1.3 *Sia $\varphi[v]$ un invariante per il ciclo* `while (`b`)` C*, e $\tau(v)$ una funzione che dipende dai valori delle variabili di programma. Se $\varphi[v] \wedge b[v]$ implica che:*

1. $\tau(v) \geq 0$
2. $\tau(v) > \tau(v')$

allora il ciclo `while (`b`)` C *termina in un numero finito di passi.*

Nell'esempio della funzione somma, consideriamo come funzione di terminazione $\tau(m, n) = n$. Come visto prima $n > 0$ durante l'esecuzione del ciclo (grazie alla guardia). E inoltre $n' = n - 1 < n$.

Osserviamo infine, che la funzione di terminazione fornisce anche un *limite superiore* (o *upper bound*) al numero di iterazioni che il ciclo esegue. Nel nostro caso, possiamo tranquillamente concludere che il numero di iterazioni che vengono eseguite nella funzione `somma` è lineare in n.

1.4 La Funzione Predecessore

Nella funzione `somma` della Sezione 1.2, abbiamo momentaneamente espresso la soluzione riferendoci a un esecutore più potente, in grado di sottrarre 1. Questo procedimento, non lo ripeteremo mai abbastanza, non solo è lecito, ma è vivamente consigliato, in quanto permette di ridurre la complessità della soluzione di un problema, riducendola alla soluzione di sottoproblemi. In generale, sarà sempre consigliato esprimere la soluzione di un problema in termini di un *esecutore astratto*, che offra al

```
public static int pred(int n){
/* REQUIRE: n>0.
 * EFFECTS: returns n-1.
 */
  int i=0;
  int j=1;
  while (j!=n) {
      /* INV: i=j-1. T=n-j. */
       i++;
       j++;
  }
  return i;
}
```

Figura 2: Funzione per calcolare il predecessore

programmatore astrazioni vicine al problema da risolvere. In un secondo momento, si andranno ad implementare le azioni di tale esecutore astratto riducendole via via fino a ricondursi alle astrazioni offerte dall'esecutore concreto (TinyJava nel nostro caso). Questa metodologia è anche spesso chiamata *top-down* (o per *raffinamenti successivi*).

Vediamo ora l'implementazione del predecessore. Vi sembri strano o no, fu il passo più duro da compiere per Alonzo Church, per formulare la celebre *Tesi di Church*[1]. C'è da dire che nel λ-calcolo (il formalismo con cui lavorava Church nel 1936), implementare il predecessore è decisamente più ostico che in TinyJava. Tuttavia l'idea è la stessa.

Innanzitutto, ricordiamo che il predecessore è definto come una funzione *parziale* $\mathsf{pred} : \mathbb{N} \to \mathbb{N}$, tale che $\mathsf{pred}(n+1) = n$. Osservate, che pred non è definita su 0, che nei naturali non è il successore di nessun numero.

L'idea per calcolare $n-1$ usando solo il successore, è quella di usare due contatori i e j, e tenere j un passo avanti ad i. Quando j raggiunge il valore n, i conterrà il valore $n-1$. In sostanza, si deve scrivere un ciclo che garantisca l'invariante $i = j-1$. Perchè l'invariante sia garantito all'inizio del ciclo è sufficiente inizializzare i a 0 e j a 1. Il comando `i=i+1; j=j+1` fa al caso nostro, in quanto in questo caso $i' - j' = (i+1) - (j+1) = i - j = 1$. Il ciclo terminerà quando $j = n$. E chiaramente dall'invariante $i = j - 1$ e dalla negazione della guardia $j = n$, otteniamo per sostituzione immediatamente l'asserzione finale desiderata, cioè $i = n - 1$. Per assicurare che l'invariante sia soddisfatto all'ingresso del ciclo, sarà

[1] La Tesi di Church dice che tutte le funzioni calcolabili sono calcolabili nel λ-calcolo. Siccome dal 1936 ad oggi, tutti i modelli di calcolo sono risultati essere equivalenti tra loro dal punto di vista dell'insieme delle funzioni calcolabili, noi la potremo rifrasare dicendo che tutte le funzioni calcolabili, sono calcolabili da un programma TinyJava. Anzi, potremo anche fare a meno dell'`if`, delle variabili booleane e scrivere programmi che usano solo due variabili!

```
public static int pred2(int n){
  int i=0;
  while (i+1!=n) i++;
  return i;
}
```

Figura 3: Funzione alternativa per calcolare il predecessore

sufficiente inizializzare i a 0 e j a 1. A questo punto, il programma si scrive da solo, ed è illustrato in Fig. 2.

La naturale funzione di terminazione è $\tau(n, i, j) = n - j$. Se n fosse 0, cioè se le precondizioni non fossero rispettate, $\tau(0, 0, 1) = -1$ non sarebbe positiva e quindi non rispetterebbe le ipotesi del Teorema 1.3. Infatti, la funzione non termina se viene passato il valore 0. Ma questo, è un problema del chiamante. Ricordiamo che, nella funzione `somma` di Fig. 1, la guardia assicura che la chiamata `pred(n)` avviene sempre con $n > 0$.

I pignoli potrebbero osservare che è in realtà sufficiente un unico contatore. Il programma è presentato in Fig. 3. Lasciamo al lettore il piacere di trovare l'invariante e dimostrare che anche questa implementazione è corretta rispetto alle specifiche.

1.4.1 Somma Reloaded

Abbiamo detto a proposito della funzione `somma` in Fig. 1 che l'iterazione principale viene eseguita n volte. Tuttavia il suo costo non è costante a causa dell'invocazione della funzione predecessore che, anch'essa ha costo lineare in n. Ne deriva che il costo computazionale della funzione `somma` si ottiene come $n + (n-1) + (n-2) + \ldots + 1$, che come tutti sapete vale $\frac{n(n+1)}{2}$, in cui il termine dominante è nell'ordine di n^2.

Credo che tuttavia, ricordando l'esperienza infantile di contare sulle dita, sia chiaro che la somma si possa fare con un numero di operazioni di successore lineare in n. Ciò si ottiene facilmente contando in avanti. Per fare ciò è sufficiente prendere un contatore, inizializzarlo a 0, incrementarlo di 1 ad ogni iterazione, e fermarsi quando raggiunge il valore di n. In Fig. 4, è mostrata la funzione corrispondente.

Vediamo rapidamente la dimostrazione di correttezza. L'invariante $s = m + i$ è soddisfatto all'ingresso del ciclo perchè s è inizializzato ad m ed i a 0. Viene mantenuto dal corpo del ciclo perchè $s = m + i$ implica ovviamente che $s + 1 = m + (i + 1)$, ma $s' = s + 1$, $i' = i + 1$ e $m' = m$, da cui otteniamo $s' = m' + i'$. Infine, $s = m + i$ assieme alla negazione della guardia $i = n$, per sostituzione implica che $s = m + n$, e quindi s è proprio il valore da tornare. Infine, l'iterazione termina perchè $n - i$ è positiva sotto le precondizioni, ed è strettamente decrescente in quanto ho $n' - i' = n - (i + 1) = n - i - 1 < n - i$.

```
public static int sommaEff(int m, int n){
/* REQUIRE: m,n>=0.
 * EFFECTS: returns m+n.
 */
  int i=0;
  int s=m;
  while (i!=n) {
      /* INV: s=m+i. T=n-i. */
       s++;
       i++;
  }
  return s;
}
```

Figura 4: Funzione "efficiente" per la somma

1.5 Problemi, Problemi... sempre Problemi

Le piccole funzioni viste finora contengono in realtà schemi di programma che facilmente si adattano alla soluzione di altri problemi. Vediamo alcuni esempi.

1.5.1 Moltiplicazione ed Esponenziale tra Naturali

Problema: *Scrivere una procedura risolutiva che calcola il prodotto tra due numeri naturali.*

Così come la somma si calcola iterando l'operazione di contare (ossia il `+1`), il prodotto si può ottenere iterando la funzione `somma`. Il corrispondente programma è mostrato in Fig. 5. Osserviamo semplicemente l'inizializzazione della variabile `p` a 0, che è l'elemento neutro della somma. Il lettore è invitato a ripercorrere la dimostrazione di correttezza fatta per la somma, facendo le opportune modifiche.

```
public static int mul(int m, int n){
/* REQUIRE: m,n>=0.
 * EFFECTS: torna m*n.
 */
 int i=0;
 int p=0;

 while (i!=n) {
     /* INV: p= m*i */
      p=somma(p,m);
      i++;
 }
 return p;
}
```

Figura 5: Funzione che calcola il prodotto

Facciamo notare che questo schema di programma è valido per molti calcoli. Ad esempio, può essere facilmente applicato per calcolare l'esponenziale, cioè m^n, semplicemente osservando che l'esponenziale è l'iterazione del prodotto (che abbiamo appena inserito tra le abilità del nostro esecutore, scrivendo la funzione `mul`). L'unica avvertenza, consiste nell'usare l'accortezza di inizializzare la variabile che conterrà il risultato finale, `e` a 1 e non a zero, cioè all'elemento neutro del prodotto. Il programma (del tutto simile al precedente) è mostrato in Fig. 6

```
public static int esp(int m, int n){
/* REQUIRE: m,n>=0.
 * EFFECTS: torna m^n
 */
 int i=0;
 int e=1;

 while (i!=n) {
     /* INV: e= m^i */
      e=mul(e,m);
      i=i+1;
 }
 return e;
}
```

Figura 6: Funzione che calcola l'esponenziale

1.5.2 Sottrazione tra Naturali

Problema: *Scrivere una procedura risolutiva che calcola la differenza tra due numeri naturali.*

Innanzitutto, osserviamo che, ancora una volta, possiamo ripetere il solito schema di programma e scrivere una funzione che itera n volte il predecessore partendo da m. Il relativo programma è mostrato in Fig. 7.

Abbiamo già notato che iterare il predecessore non è una buona idea in quanto porta a un programma di complessità quadratica. D'altra parte, è possibile semplicemente (contando in avanti) contare quanti passi servono per andare da n ad m, ottenendo una funzione di complessità $m - n$. Il codice relativo è mostrato in Fig. 8.

In entrambi i casi, le funzioni non terminano se $n > m$. Nel primo caso perchè andremo ad invocare la funzione `pred` su un parametro negativo, e nel secondo caso perchè contando in avanti a partire da un numero maggiore di m, non si può mai raggiungere m e quindi la condizione di uscita dal ciclo non si verifica mai.

Questo comportamento non vi sconcerti. Sui numeri naturali, la sottrazione non è un'operazione interna, e non è definita quando il secondo parametro è maggiore del secondo. Chiaramente, volendo evitare la non terminazione, dovremmo mettere un controllo prima di cominciare il ciclo. Vedremo più avanti nel corso, come si

```
public static int diff(int m, int n){
/* REQUIRE: m>=n>=0.
 * EFFECTS: returns m-n.
 */
  int i=0;
  int d=m;
  while (i!=n) {
      /* INV: d=m-i */
       d=pred(d);
       i++;
  }
  return d;
}
```

Figura 7: Funzione che calcola la differenza iterando il predecessore

```
public static int diffEff(int m, int n){
/* REQUIRE: m>=n>=0.
 * EFFECTS: returns m-n.
 */
  int d=0;
  int i=n;

  while (i!=m) {
      /* INV: n+d=i */
       d++;
       i++;
  }
  return d;
}
```

Figura 8: Funzione efficiente per il calcolo della differenza

tratta in modo maturo la situazione in cui i parametri di ingresso non soddisfino le precondizioni in un linguaggio di programmazione sofisticato come JAVA, quando parleremo di *eccezioni*. Per ora accontentiamoci di dire che la nostra funzione soddisfa le specifiche.

1.5.3 Confronto tra Naturali

Problema: *Scrivere una procedura risolutiva che prende in ingresso due parametri m ed n e restituisce 1 se $m < n$, 0 se $m = n$, e -1 se $m > n$.*

Mentre nella differenza potevamo allegramente fare l'ipotesi che il secondo parametro fosse minore del primo, qui, il nostro compito è proprio determinare quale dei due parametri sia minore dell'altro. Potendo fare solo il successore, come sempre conviene contare a partire da 0 e continuare fino a che non si incontra uno dei numeri ricevuti in ingresso. Se si incontra m si restituisce 1, se si incontra n si resti-

tuisce -1. Per mantenere il programma semplice, conviene liberarsi subito del caso $m = n$: lo possiamo fare agevolmente, perchè possiamo usare il test di uguaglianza. La funzione `compare` è presentata in Fig. 9. Lasciamo, ancora una volta al lettore la semplice verifica che la proprietà $i \leq m \wedge i \leq n$ è una proprietà invariante per il ciclo e implica la correttezza della funzione.

```
public static int compare(int m, int n){
/* REQUIRE: m,n>=0.
 * EFFECTS: if m<n returns 1, if n==m returns 0, else returns -1.
 */
  int i=0;
  if (n==m) return 0;
  while (true) {
      /* INV i<=n,m */
       if (i==n) return -1;
       if (i==m) return 1;
       i++;
   }
}
```

Figura 9: Funzione efficiente calcolo della differenza

Osservate che, a questo punto, possiamo scrivere la funzione che calcola il minore uguale, `minUg`, semplicemente invocando la funzione `compare`, come segue:

```
public static boolean minUg(int m, int n){
    int c=compare(m,n);
    if (c==0) return true;
    if (c==1) return true;
    return false;
```

1.6 Progettare con le Asserzioni: Divisione Intera

Chi avesse avuto la fortuna di assistere a una lezione del prof. Edsgar W. Dijkstra[2] lo avrebbe visto scrivere vicino al margine inferiore della lavagna l'asserzione finale che il programma avrebbe dovuto soddisfare e procedere a ritroso scrivendo comandi, fintantochè sulla lavagna non sarebbe apparsa l'asserzione iniziale (cioè le precondizioni) del programma.

In questa sezione vorrei appunto mostrarvi come le asserzioni logiche, oltre a uno strumento per descrivere le specifiche di una funzione (e quindi aiutare la comunicazione tra più programmatori per il suo corretto utilizzo), oltre a uno strumento per dimostrarne formalmente la correttezza, possono essere anche lo strumento che *guida la stesura di un programma*. Ancora una volta, faremo questo percorso seguendo un esempio.

[2]con C. A. R. Hoare, Dijkstra sviluppò la teoria delle asserzioni logiche e numerosissimi sono i suoi contributi alla teoria e alla pratica dei linguaggi di programmazione e degli algoritmi.

Problema: *Scrivere una procedura risolutiva che calcola quoziente e resto della divisione tra due numeri naturali.*

Cominciamo formalizzando rigorosamente il problema della divisione intera tra due numeri naturali m ed n: bisogna trovare due numeri interi q (quoziente) ed r (resto) per i quali valga la seguente asserzione:

$$m = qn + r \ \wedge \ 0 \leq r < n$$

che è quindi la naturale asserzione finale del programma che dobbiamo scrivere. Osserviamo che la relazione $\phi_1 \equiv m = qn + r$ è soddisfatta per un'infinità di coppie di naturali q ed r, mentre la richiesta $\phi_2 \equiv 0 \leq r < n$, rende le soluzioni uniche (geometricamente, le possibili soluzioni di ϕ_1 sono rappresentabili con una retta sul piano cartesiano). Quindi, partendo da una qualsiasi coppia r_0, q_0 che soddisfa ϕ_1 e "camminando" sulla retta di equazione $r = -qn + m$ nella direzione giusta, raggiungiamo la coppia r, q desiderata. Questo ragionamento ci suggerisce che l'invariante della nostra iterazione sarà appunto ϕ_1, mentre la condizione d'uscita sarà $0 \leq r < n$. Avendo una coppia q, r che soddisfa ϕ_1, ne possiamo ottere un'altra incrementando (o decrementando) q di 1 e contemporaneamente decrementando (o incrementando) r di n. Una coppia che soddisfa banalmente ϕ_1 è chiaramete $q_0 = 0$ ed $r_0 = m$. Siccome q dovrà essere un numero positivo, cammineremo sulla retta incrementando q e decrementando r. Il comando {`q=q+1; r=dif(r,n)`} fa chiaramente al caso nostro per mantenere l'invariante ϕ_1.

Il programma risultante, è mostrato in Fig. 10. Otteniamo gratis anche il programma che calcola il resto della divisione intera, mostrato in Fig. 11.

```
public static int quoziente(int m, int n){
/* REQUIRE: m>=0,n>0.
 * EFFECTS: returns q: Exists r. q*n+r=m & r<n.
 */
 int q=0;
 int r=m;

 while (minUg(n,r)) {
     /* INV: m = q*n + r. T=r. */
      r=difEff(r,n);
      q=q+1;
 }
 return q;
}
```

Figura 10: Funzione che calcola il quoziente della divisione intera

La terminazione è garantita dalla funzione $\tau(m, n, q, r) = r$. r_0 infatti è positivo, e sotto la condizione $n \geq r$ (garantita dalla guardia del ciclo) anche $r' = r - n$ è ancora positivo. Inoltre, essendo per le precondizioni $n > 0$, ho che $r' < r$.

```
public static int resto(int m, int n){
/* REQUIRE: m>=0,n>0.
 * EFFECTS: returns r: Exists q: q*n+r=m & r<n
 */
 int q=0;
 int r=m;

 while (minUg(n,r)) {
     /* INV: m = q*n + r. T=r. */
      r=difEff(r,n);
      q=q+1;
 }
 return r;
}
```

Figura 11: Funzione che calcola il resto della divisione intera

Esercizi e Spunti di Riflessione

1. Scrivere una funzione per il prodotto tra due numeri naturali che non invoca la funzione `somma`, ma che viene scritta diretamente in termini del successore. Osservare come la soluzione sia molto più complessa e difficile da scrivere e analizzare.

2. Provate a calcolare la complessità delle funzioni `mul` e `esp` in termini del numero di volte che viene eseguita una operazione di successore. Cosa ne deducete?

3. Scrivere una funzione `pari(int n)` che restituisca `true` se la variabile `n` contiene un numero pari, e `false` altrimenti. Provare a dare una soluzione che non fa uso della funzione `resto`.

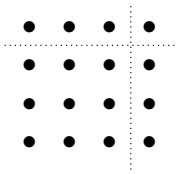
4. Scrivere una funzione che calcola la somma dei primi n numeri.

5. Scrivere una funzione che calcola il quadrato di un numero n come somma dei primi n numeri dispari. Infatti:

   $$1^2 = 1 = 1 \qquad 2^2 = 4 = 1 + 3 \qquad 3^2 = 9 = 1 + 3 + 5$$

   e così via.

   Questo problema ha un interessante interpretazione geometrica:



(Per passare dal quadrato di lato n a quello di lato $n+1$ bisogna aggiungere due file di n pallini, più un ultimo pallino, cioè $2n+1$ pallini).

6. Scrivere una funzione che calcola la radice quadrata intera di n, cioè trovi un numero r, tale che $r^2 \leq n < (r+1)^2$.

7. Scrivere una funzione che calcola il *più grande* numero primo che divide n. Scrivere precondizioni e postcondizioni. Corredare eventuali comandi iterativi con invariante. Dimostrare la correttezza della funzione.

1.7 ⋆Fuori Programma: Triple di Hoare

Per amore di completezza e per gli studenti interessati e curiosi, propongo questa sezione in cui viene descritto brevemente come sia possibile *formalizzare* ulteriormente la teoria presentata sopra con un sistema di prova che permette di effettuare dimostrazioni formali (nel senso di sistemi logici come la *deduzione naturale* che dovreste aver visto in un corso base di Matematica per l'Informatica). La formalizzazione rigorosa dei ragionamenti sulle asserzioni logiche avviene attraverso regole di deduzione che specificano come i programmi possano soddisfare una certa asserzione logica. Ciò permette di dare dimostrazioni formali, e di conseguenza verificabili *meccanicamente* (cioè da un calcolatore). L'idea è quella di provare enunciati del tipo:

$$\psi \quad \texttt{C} \quad \varphi$$

che hanno il seguente significato intuitivo: se ψ è una asserzione logica soddisfatta dallo stato (ψ dipende in genere da valori assunti dalle variabili usate dal programma), allora l'asserzione φ sarà soddisfatta nello stato ottenuto dopo l'esecuzione del comando `C`, nel caso in cui `C` *termini*. Le regole di deduzione sono date per induzione sulla *struttura sintattica* dei comandi. Osservate che, la regola del `while` è giustificata dai teoremi visti in Sezione 1.3. Osservate bene anche la regola dell'assegnazione, che, a prima vista, potrebbe sembrare al quanto controintuitiva.

$$\frac{\varphi\ \{\texttt{C}\}\ \psi \quad \psi\ \{\texttt{D}\}\ \phi}{\varphi\ \{\texttt{C;D}\}\ \phi}\ (\textsc{Sequenza}) \qquad \frac{\varphi \wedge b\ \{\texttt{C}\}\ \varphi \quad \varphi \wedge \neg b \Rightarrow \psi}{\varphi\ \{\texttt{while(b) C}\}\ \psi}\ (\textsc{While})$$

$$\frac{\varphi \wedge b\ \{\texttt{C}\}\ \psi \quad \varphi \wedge \neg b\ \{\texttt{D}\}\ \psi}{\varphi\ \{\texttt{if (b) C else D}\}\ \psi}\ (\textsc{If}) \qquad \frac{}{\varphi[E/x]\ \{\texttt{x = E}\}\ \varphi[x]}\ (\textsc{Assegnazione})$$

Figura 12: Triple di Hoare (Regole fondamentali)

```
int multiplyingLikeAnEgiptian(int m, int n){
/* PREC: m,n>= 0 */
    int p=0;
    while (n!=0){
    /* INV: m0 * n0 = p+m*n  */
       if (resto(n,2) == 0)
             { m = somma(m,m);
               n = div(n,2);
               }
         else
             { p = somma(p,m);
               n = pred(n);
               }
      }
    return p;
}
```

Figura 13: Funzione per la moltiplicazione egiziana

2 Alcuni Esercizi Svolti

Quest'ultima sezione è dedicata a qualche esempio un po' più impegnativo dal punto di vista algoritmico.

2.1 Multiplying like an Egyptian

Problema *Scrivere una funzione che calcoli la moltiplicazione sfruttando le seguenti uguaglianze:*

$$\begin{aligned} m \times 2n &= (m+m) \times n \quad (n > 1) \\ m \times (2n+1) &= m \times 2n + m \\ m \times 0 &= 0 \end{aligned}$$

Questo algoritmo (che si può applicare anche al calcolo dell'esponenziale) si basa sull'idea che moltiplicare per 2 o fare somme è relativamente semplice rispetto a fare prodotti. È noto come *moltiplicazione egiziana*, in quanto sembra fosse già noto agli antichi egizi. Per la verità qualcuno lo chiama *moltiplicazione russa*. Il suo grande vantaggio, rispetto alle n somme richieste dalla funzione in Fig. 5, è che permette di fare un numero di somme molto piccolo (quante?).

Osserviamo, per i non esperti di matematica, che la definizione di moltiplicazione è data per *induzione* sul secondo fattore. Viene definito il significato della moltiplicazione distinguendo i casi in cui il secondo fattore sia un numero pari (maggiore di zero!), un numero dispari oppure zero (caso base). In ogni caso, il significato di $m \times n$ $(n > 1)$ viene dato induttivamente in termini di una moltiplicazione con un secondo fattore minore.

```
public static int mcdIngenuo(int m, int n){
/* REQUIRE: m,n>0.
 * EFFECTS: returns MCD(m,n).
 */
 int mcd=min(m,n);
 while (!divide(mcd,m) || !divide(mcd,n)){
     /* INV: forall k>mcd. !k|m || !k|n */
      mcd=pred(mcd);
 }
 return mcd;
}
```

Figura 14: Funzione che calcola l'MCD tra due numeri (algoritmo "ingenuo")

Soluzione: L'idea di fare solo le operazioni "semplici" (cioè somme e predecessori) accumulando i risultati in una variabile `p`, in modo che venga mantenuta la proprietà invariante $m_0 n_0 = mn + p$. Questa relazione è banalmente soddisfatta all'ingreso del ciclo inizializzando `p` a 0. Quando n è pari, raddoppiando m e dividendo n per 2, la relazione si mantiene perchè $m'n' + p' = 2m\frac{n}{2} + p = mn + p$. Quando n è dispari, toglieremo 1 a n e aggiungeremo m a p, mantenendo la relazione in quanto in questo caso $m'n' + p' = m(n-1) + (p+m) = mn - m + p + m = mn + p$. Usciremo dal ciclo quando raggiungiamo il caso base delle equazioni induttive scritte sopra, cioè quando $n = 0$. In tale situazione $mn + p = m_0 n_0$ implica proprio che $p = m_0 n_0$, che è quanto volevamo ottenere. Il caso base viene raggiunto, perchè la guardia ($n \neq 0$) implica che $n' < n$. Fatte queste osservazioni, il programma si scrive da solo, come in Fig. 13.

2.2 La Maestra ed Euclide

Problema: *Scrivere una funzione che calcola il massimo comun divisore tra due numeri.*

Soluzione 1: Per risolvere questo problema seguiremo diverse strategie. Cominciamo con un algoritmo "ingenuo". Siano m ed n i numeri di cui vogliamo calcolare $\mathsf{mcd}(m, n)$. Prendiamo il minore dei due (diciamo sia $p = \mathsf{min}(m, n)$) e proviamo tutti i numeri $p, p-1, p-2 \ldots 2, 1$. Il primo valore che divide sia m che n è il massimo comun divisore di m ed n. In Fig. 14 trovate il programma corrispondente. Osservate che la correttezza di questa funzione è implicata dall'invariante $\forall k > mcd. \neg(k \mid m) \vee \neg(k \mid n)$, dove $k \mid n$ significa "k divide n".

Abbiamo usato, per comodità l'operatore logico `||` che fa l'or logico tra due condizioni. Esiste, anche un operatore per l'and, che si scrive `&&`. Il lettore è invitato a verificare che si possono facilmente scrivere due funzioni `or` e `and` che dimostrano che anche questi operatori non sono necessari, ma possono essere definiti in TinyJava (a meno di alcune precisazioni che faremo nel seguito del corso). Abbiamo

anche usato la funzione booleanda `divide`. Essa può essere facilmente implementata a partire dalla funzione `resto`, un po' come `minUg` è stata implementata usando `compare`. L'uso della condizione `!divide(mcd,m)` al posto di `resto(m,mcd)!=0` è semplicemente motivato da un'esigenza di *chiarezza*. Visto che spesso i programmi devono essere modificati è buona norma abituarsi fin da subito a scriverli nel modo più chiaro possibile e più vicino possibile al linguaggio del problema in esame.

Soluzione 2: Il metodo precedente, in effetti, benchè intollerabile per un esecutore umano, non è poi così male per i calcolatori moderni, che riescono a fare moltissime operazioni semplici in poco tempo. Certo, dovessimo chiederci chi è il massimo comun divisore tra due numeri di 10 cifre ciascuno, primi tra loro, il costo computazionale diventerebbe proibitivo. Un algoritmo più intelligente è quello che vi è noto già dalle elementari (o medie?). La maestra recitava una filastrocca del tipo:

> *Il massimo comun divisore di due numeri è il prodotto di tutti i fattori (primi) comuni, presi con il loro minimo esponente.*

Lasciamo al lettore il piacere di dimostrare che questa filastrocca corrisponde a una verità matematica. Ma come insegnare questa filastrocca al nostro calcolatore usando TinyJava? Il compito sembra a prima vista proibitivo: infatti il numero di fattori di un certo naturale n è variabile (e potenzialmente illimitato al crescere di n). Allo stato delle nostre conoscenze, avremmo bisogno di un numero non prevedibile a priori di variabili.

Risolviamo dapprima un compito più semplice. Scriviamo una funzione che scrive tutti i fattori primi di un numero n (Fig. 15). Anche qui, ci sono molte questioni interessanti da analizzare. Perchè ad esempio, tale programma stampa solo i fattori primi? Cercate almeno una giustificazione informale, oppure sostituite opportunamente i `??` nell'invariante del ciclo. I pigri possono sempre continuare indolentemente la lettura, sperando che questi misteri siano (parzialmente) svelati nel seguito.

Facciamo inoltre conoscenza con una particolare forma di funzione, in cui il valore di ritorno è `void`. `void` è un curioso tipo, che ha un solo elemento `()`. In generale, viene dichiarato `void` il tipo di ritorno delle funzioni che non restituiscono nessun valore. E infatti, la nostra funzione non ha un'istruzione di `return`. Potrebbe in realtà avercela, ma senza un valore da tornare, al solo scopo di interrompere prematuramente l'esecuzione della funzione.

Tali funzioni, tradizionalmente vengono dette *procedure*. Idealmente, una funzione è un'astrazione di un'espressione, mentre una procedura è un'astrazione di un comando. Le procedure infatti, usualmente, modificano lo stato *globale* della memoria (comportandosi quindi come comandi), oppure come nel nostro caso fanno dell'input/output.

```
public static void stampaDivisoriPrimi(int m){
/* REQUIRE: n>0.
 * EFFECTS: stampa tutti i divisori primi di n.
 */
    int p=2;

    while (p*p<=m){
    /* INV: ??*/
       if (!divide(m,p)) p++;
          else { System.out.println(p);
                 m = quoziente(m,p);
                 }
       }
    if (m!=1) System.out.println(m);
    }
```

Figura 15: Procedura che stampa i fattori primi di un numero

Torniamo al nostro problema. Il programma in Fig. 15 genera i fattori primi di un numero in *ordine crescente*. Ma per risolvere il nostro problema è veramente necessario memorizzarli? La risposta è ovviamente no. Basta generare i fattori primi di entrambi i numeri in parallelo (con la dovuta cura) e accumulare i prodotti dei fattori primi che dividono entrambi i numeri. Dobbiamo distinguere con cura i casi in cui troviamo un primo che divide entrambi i numeri, nessuno dei due numeri, o solo uno dei due. Lasciamo al lettore il piacere di verificare che la proprietà $\mathsf{mcd}(m_0, n_0) = mcd \times \mathsf{mcd}(m, n)$ è effettivamente un invariante.

Per quanto riguarda la terminazione, quando la guardia è verificata, si ha chiaramente $m, n \geq p$ e quindi anche che la quantità $m + n - p$ è positiva. Inoltre, ad ogni iterazione o viene incrementato p o almeno uno tra m ed n viene diviso per $p > 1$. Quindi, necessariamente si ha $m' + n' - p' < m + n - p$.

Soluzione 3: Già gli antichi greci avevano osservato che se p divide m e n, allora divide anche $m - n$ (o $m + n$). Questa osservazione subito suggerisce le seguenti equazioni induttive (tradizionalmente attribuite ad Euclide) per definire il massimo comun divisore $\mathsf{mcd}(m, n)$ di due numeri strettamente positivi.

$$\begin{array}{rcll} \mathsf{mcd}(n, n) & = & n & \\ \mathsf{mcd}(m, n) & = & \mathsf{mcd}(m - n, n) & (n < m) \\ \mathsf{mcd}(m, n) & = & \mathsf{mcd}(n - m, m) & (m < n) \end{array}$$

Analogamente a quanto fatto per la moltiplicazione egiziana, queste equazioni si traducono abbastanza naturalmente nel programma C in Fig. 17, che rispetto ai precedenti, spicca in eleganza, semplicità ed efficienza:

```
public static int mcdMaestra(int m, int n){
/* REQUIRE:m,n>0.
 * EFFECTS: returns MCD(m,n).
 */
 int mcd=1;
 int p=2;
 int q1,q2,r1,r2;
 while (minUg(p,min(m,n))){
     /* INV: forall k<p !k|m && !k|n && MCD(m0,n0)=mcd*MCD(m,n) */
      boolean dpm,dpn;
      dpm=divide(p,m);
      dpn=divide(p,n);
      if (dpm) m=quoziente(m,p);
      if (dpn) n=quoziente(n,p);
      if (!dpn && !dpm) p++;
      if (dpm && dpn) mcd=mult(mcd,p);
  }
  return mcd;
}
```

Figura 16: Procedura che calcolo l'mcd con l'algoritmo insegnato a scuola

Esercizi e Spunti di Riflessione

1. Adattare la funzione della moltiplicazione egiziana per calcolare l'esponenziale.
2. Trovare 3 funzioni che calcolano il *minimo comune multiplo*, ciascuna delle quali sia la corrispondente di una delle 3 funzioni che abbiamo considerato per il massimo comun divisore.
3. Scrivere una funzione che scrive tutte le coppie che danno come prodotto un certo numero naturale n.
4. ★Scrivere una funzione che scrive tutti i possibili prodotti che danno come risultato un numero naturale n.

```
public static int mcdEuclide(int m, int n){
/* REQUIRE: m,n>0.
 * EFFECTS: returns MCD(m,n).
 */
 while (m!=n){
     /* INV: MCD(m,n)=MCD(m0,n0) */
      if (minUg(n,m)) m=diff(m,n);
          else n=diff(n,m);
 }
 return n;
}
```

Figura 17: mcd calcolato con l'algoritmo di Euclide