Fondamenti di Informatica (Elettronici)

THINK JULIA – Capitolo 6

3 novembre 2020

6. Funzioni feconde[1]

Molte delle funzioni Julia che abbiamo utilizzato, come le funzioni matematiche, producono valori di ritorno.

Ma le funzioni che abbiamo scritto sono tutte vuote: hanno un effetto, come stampare un valore o spostare una tartaruga, ma restituiscono "`nothing`".

In questo capitolo impareremo a scrivere funzioni fruttuose (o feconde).

Section 1

Valori restituiti

`nothing` come valore di ritorno

La chiamata della funzione genera un valore di ritorno, che di solito assegniamo a una variabile o utilizziamo come parte di un'espressione.

```
e = exp(1.0)
height = radius * sin(radians)
```

Le funzioni che abbiamo scritto finora sono nulle.

Parlando semplicemente, non hanno valore di ritorno; più precisamente, il loro valore di ritorno è `nothing`.

In questo capitolo, scriveremo (finalmente) funzioni feconde. Il primo esempio è `area`, che restituisce l'area di un cerchio con il raggio dato.

Valore "return" implicito

Abbiamo già visto l'istruzione `return` in precedenza, ma in una funzione fruttuosa l'istruzione `return` include un'espressione. Questa istruzione significa: "Torna immediatamente da questa funzione e utilizza la seguente espressione come valore di ritorno."

L'espressione può essere arbitrariamente complicata, quindi avremmo potuto scrivere questa funzione in modo più conciso:

```
function area(radius)
    pi * radius^2
end
```

Il valore restituito da una funzione è il valore dell'ultima espressione valutata, che, per definizione, è l'ultima espressione nel corpo della funzione.

Più istruzioni "return"

D'altra parte, variabili temporanee e dichiarazioni di ritorno esplicite possono semplificare il debug.

A volte è utile avere più istruzioni di ritorno, una in ogni ramo di un condizionale:

```
function absvalue(x)
    if x < 0
        return -x
    else
        return x
    end
end
```

Poiché queste istruzioni `return` sono in un condizionale alternativo, solo una viene eseguita.

Molteplici percorsi di controllo

Non appena viene eseguita un'istruzione `return`, la funzione termina senza eseguire alcuna istruzione successiva.

Il codice che appare dopo un'istruzione `return`, o qualsiasi altro punto in cui il flusso di esecuzione non potrà mai raggiungere, è chiamato codice morto.

In una funzione feconda, è una buona idea assicurarsi che ogni possibile percorso attraverso il programma raggiunga un'istruzione `return`.

```
function absvalue(x)
    if x < 0
        return -x
    end
    if x > 0
        return x
    end
end
```

Valore assoluto

Questa funzione non è corretta perché se `x` è 0, nessuna delle due condizioni è verae la funzione termina senza raggiungere un'istruzione `return`.

Se il flusso di esecuzione arriva alla fine di una funzione, il valore restituito è "`nothing`", che non è il valore assoluto di 0.

```
julia> show(absvalue(0))
nothing
```

Valore assoluto

Questa funzione non è corretta perché se `x` è 0, nessuna delle due condizioni è verae la funzione termina senza raggiungere un'istruzione `return`.

Se il flusso di esecuzione arriva alla fine di una funzione, il valore restituito è "`nothing`", che non è il valore assoluto di 0.

```
julia> show(absvalue(0))
nothing
```

SUGGERIMENTO

Julia fornisce una funzione predefinita chiamata `abs` che calcola i valori assoluti.

Section 2

Sviluppo incrementale

Esempio

Con funzioni più grandi, si può dedicare più tempo al debug

Per gestire programmi sempre più complessi, possiamo utilizzare un processo chiamato sviluppo incrementale.

L'obiettivo dello sviluppo incrementale è evitare lunghe sessioni di debug, aggiungendo e testando solo una piccola quantità di codice alla volta.

Ad esempio, supponiamo di voler trovare la distanza tra due punti, di coordinate $(x1, y1)$ e $(x2, y2)$.

Per il teorema di Pitagora, la distanza è:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Il primo passo è considerare come dovrebbe apparire una funzione di distanza in Julia. In altre parole, quali sono gli input (parametri) e qual è l'output (valore di ritorno)?

In questo caso, gli input sono due punti, che puoi rappresentare utilizzando quattro numeri. Il valore restituito è la distanza, rappresentata da un valore in virgola mobile.

Schema delle funzioni

All'inizio, si può scrivere uno schema della funzione:

```
function distance(x_1, y_1, x_2, y_2)
    0.0
end
```

Ovviamente, questa versione non calcola le distanze; restituisce sempre zero.

Ma è sintatticamente corretta e funziona, il che significa che puoi testarla prima di renderla più complicata. I numeri in pedice sono disponibili nella codifica dei caratteri Unicode (`_1 TAB`, `_2 TAB`, ecc.).

Per testare la nuova funzione, chiamala con argomenti di esempio:

```
distance(1, 2, 4, 6)
```

Ho scelto questi valori in modo che la distanza orizzontale sia 3 e la distanza verticale sia 4; in questo modo, il risultato è 5, l'ipotenusa di un triangolo 3-4-5.

Quando si testa una funzione, è utile conoscere la risposta giusta.

Esempio incrementale

A questo punto abbiamo confermato che la funzione è sintatticamente corretta e possiamo iniziare ad aggiungere codice al corpo.

Un passo successivo ragionevole è calcolare le differenze "`x2 - x1`" e "`y2 - y1`". La prossima versione memorizza questi valori in variabili temporanee e li stampa con la macro @show.

```
function distance(x1, y1, x2, y2)
    dx = x2 - x1
    dy = y2 - y1
    @show dx dy
    0.0
end
```

Se la funzione funziona, dovrebbe visualizzare "`dx = 3`" e "`dy = 4`". Se è così, sappiamo che la funzione sta ottenendo gli argomenti giusti e sta eseguendo correttamente il primo calcolo. In caso contrario, ci sono solo poche righe dacontrollare**.

Versione finale

Successivamente calcoliamo la somma dei quadrati di `dx` e `dy`:

```
function distance(x1, y1, x2, y2)
    dx = x2 - x1
    dy = y2 - y1
    d2 = dx^2 + dy^2
    @show d2
    return sqrt(d2)
end
```

Di nuovo, dovremmo eseguire il programma in questa fase e controllare l'output (che dovrebbe essere 25). Infine, usare `sqrt` per calcolare e restituire il risultato:

Se funziona correttamente, avremo finito. Altrimenti, potremmo voler stampare il valore di `sqrt(d2)` prima dell'istruzione `return`.

Aspetti chiave del processo

La versione finale della funzione non mostra nulla quando viene eseguita; restituisce solo un valore. Le istruzioni di stampa che abbiamo scritto sono utili per il debug, ma una volta che la funzione funziona, occorre rimuoverle.

Un codice del genere è chiamato `scaffolding` perché è utile per la creazione del programma ma non fa parte del prodotto finale.

All'inizio, dovremmo aggiungere solo una o due righe di codice alla volta. Man mano che acquisisci più esperienza, potresti ritrovarti a scrivere e eseguire il debug di blocchi più grandi. In ogni caso, lo sviluppo incrementale può far risparmiare molto tempo per il debug.

Gli aspetti chiave del processo sono:

1. Iniziare con un programma funzionante e apportare piccole modifiche incrementali. In qualsiasi momento, se c'è un errore, si avrà una buona idea di dove si trova.

#. Una volta che il programma sta funzionando, rimuovere* alcune delle impalcature* o consolidare più istruzioni in espressioni composte, ma solo se non rende il programma

Aspetti chiave del processo

La versione finale della funzione non mostra nulla quando viene eseguita; restituisce solo un valore. Le istruzioni di stampa che abbiamo scritto sono utili per il debug, ma una volta che la funzione funziona, occorre rimuoverle.

Un codice del genere è chiamato `scaffolding` perché è utile per la creazione del programma ma non fa parte del prodotto finale.

All'inizio, dovremmo aggiungere solo una o due righe di codice alla volta. Man mano che acquisisci più esperienza, potresti ritrovarti a scrivere e eseguire il debug di blocchi più grandi. In ogni caso, lo sviluppo incrementale può far risparmiare molto tempo per il debug.

Gli aspetti chiave del processo sono:

1. Iniziare con un programma funzionante e apportare piccole modifiche incrementali. In qualsiasi momento, se c'è un errore, si avrà una buona idea di dove si trova.
2. Utilizzare variabili per contenere valori intermedi in modo da poterli visualizzare e controllare.

#. Una volta che il programma sta funzionando, rimuovere* alcune delle impalcature* o consolidare più istruzioni in espressioni composte, ma solo se non rende il programma difficile da leggere

Section 3

Composizione

Composizione

Come dovremmo aspettarci, possiamo chiamare una funzione dall'interno di un'altra.

Ad esempio, scriveremo una funzione che prende due punti, il centro del cerchio e un punto sul perimetro, e calcola l'area del cerchio.

Supponiamo che il punto centrale sia memorizzato nelle variabili "`xc`" e "`yc`" e che il punto perimetrale sia in "`xp`" e "`yp`".

Il primo passo è trovare il "raggio" del cerchio, che è la distanza tra i due punti. Abbiamo appena scritto una funzione, `distance`, che fa questo:

```
radius = distance(xc, yc, xp, yp)
```

Il passo successivo è trovare l'`area` di un cerchio con quel raggio:

```
result = area(radius)
```

Incapsulamento

"Incapsulando" questi passaggi in una singola funzione, otteniamo:

```
function circlearea(xc, yc, xp, yp)
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
end
```

Le variabili temporanee `radius` e `result` sono utili per lo sviluppo e il debugging, ma una volta che il programma funziona, possiamo renderlo più conciso componendo le chiamate di funzione:

```
function circlearea(xc, yc, xp, yp)
    area(distance(xc, yc, xp, yp))
end
```

Section 4

Funzioni booleane

Funzioni predicato

Le funzioni possono restituire valori booleani, il che è spesso conveniente per nascondere test complicati all'interno delle funzioni. Per esempio:

```
function isdivisible(x, y)
    if x % y == 0
        return true
    else
        return false
    end
end
```

È comune dare nomi di funzioni booleane che suonano come domande sì / no; `isdivisible` restituisce "`true`" o "`false`" per indicare se "`x`" è divisibile per "`y`":

```
julia> isdivisible(6, 4)
false
julia> isdivisible(6, 3)
true
```

Esempi

Il risultato dell'operatore == è un booleano, quindi possiamo scrivere la funzione in modo più conciso restituendola direttamente:

```
function isdivisible(x, y)
    x % y == 0
end
```

Le funzioni booleane vengono spesso utilizzate nelle istruzioni condizionali:

```
if isdivisible(x, y)
    println("x is divisible by y")
end
```

Potremmo essere tentati di scrivere qualcosa come:

```
if isdivisible(x, y) == true
    println("x is divisible by y")
end
```

Ma il confronto extra con "true" non è necessario.

Section 5

Più ricorsione

il sottoinsieme Julia visto finora è Turing-completo

Abbiamo trattato solo un piccolo sottoinsieme di "Julia", ma potresti essere interessato a sapere che questo sottoinsieme è un linguaggio di programmazione completo, il che significa che tutto ciò che può essere calcolato può essere espresso in questo linguaggio.

Qualsiasi programma mai scritto potrebbe essere riscritto utilizzando solo le funzionalità del linguaggio che hai appreso finora (in realtà, avresti bisogno di alcuni comandi per controllare dispositivi come il mouse, i dischi, ecc., ma questo è tutto).

Dimostrare questa affermazione è un esercizio non banale realizzato per la prima volta da Alan Turing, uno dei primi scienziati informatici (alcuni sostengono che fosse un matematico, ma molti dei primi scienziati informatici hanno iniziato come matematici).

Di conseguenza, è conosciuta come la Tesi di Turing.

Per una discussione più completa (e accurata) della Tesi di Turing, raccomando il libro di Michael Sipser Introduzione alla teoria del calcolo.

Per darti un'idea di cosa puoi fare con gli strumenti che hai imparato finora, valuteremo alcune funzioni matematiche definite ricorsivamente.

Una definizione ricorsiva è simile a una definizione circolare, nel senso che la definizione contiene un riferimento alla cosa da definire. Una definizione veramente circolare non è molto utile.

Parole recorsive ;-)

vorpal

Un aggettivo usato per descrivere qualcosa che è vorpal.

Parole recorsive ;-)

vorpal

Un aggettivo usato per descrivere qualcosa che è vorpal.

GNU's Not Unix

Il sistema operativo GNU è un sistema software libero completo, compatibile con Unix. GNU sta per "GNU's Not Unix".

Definizione del fattoriale

Se cerchiamo la definizione della funzione fattoriale, indicata con il simbolo !, potremmo ottenere qualcosa del genere:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Questa definizione dice che il fattoriale di 0 è 1 e il fattoriale di qualsiasi altro valore, n, è n moltiplicato per il fattoriale di $n - 1$.

Quindi 3! è 3 per 2!, che è 2 per 1!, che è 1 per 0!. Mettendo tutto insieme, 3! è uguale a 3 per 2 per 1 per 1, che è 6.

vfill pause

Definizione induttiva In matematica e informatica, una definizione ricorsiva o induttiva viene utilizzata per definire gli elementi in un insieme in termini di altri elementi nell'insieme.

Caso base

Se possiamo scrivere una definizione ricorsiva di qualcosa, possiamo scrivere un programma Julia per valutarla. Il primo passo è decidere quali dovrebbero essere i parametri. In questo caso dovrebbe essere chiaro che fattoriale prende un numero intero:

```
function fact(n)
end
```

Se l'argomento è 0, tutto ciò che dobbiamo fare è restituire 1:

```
function fact(n)
    if n == 0
        return 1
    end
end
```

Definizione ricorsiva

Altrimenti, e questa è la parte interessante, dobbiamo fare una chiamata ricorsiva per trovare il fattoriale di `n-1` e poi moltiplicarlo per `n`:

```
function fact(n)
    if n == 0
        return 1
    else
        recurse = fact(n-1)
        result = n * recurse
        return result
    end
end
```

Una riscrittura più compatta e leggibile:

```
function fact(n)
    if n == 0 return 1
    else n * fact(n-1) end
end
```

Flusso di esecuzione

Il flusso di esecuzione di questo programma è simile al flusso del conto alla rovescia in Ricorsione.

Se chiamiamo `fact` con il valore 3:

Since `3` is not `0`, we take the second branch and calculate the factorial of `n-1` ...

Since `2` is not `0`, we take the second branch and calculate the factorial of `n-1` ...

Since `1` is not `0`, we take the second branch and calculate the factorial of `n-1` ...

Since `0` equals `0`, we take the first branch and return `1` without making any more recursive calls.

The return value, `1`, is multiplied by `n`, which is `1`, and the `result` is returned.

The return value, `1`, is multiplied by `n`, which is `2`, and the `result` is returned.

The return value `2` is multiplied by `n`, which is `3`, and the result, `6`, becomes the return value of the function call that started the whole process.

{#

.class width = 100%}

Julia fornisce la funzione "`factorial`" per calcolare il fattoriale di un intero.

Section 6

Atto di fede

Assumendo la correttezza

Seguire il flusso di esecuzione è un modo per leggere i programmi, ma può diventare rapidamente opprimente.

Un'alternativa è quello che io chiamo il "salto della fede".

Quando si arriva a una chiamata di funzione, invece di seguire il flusso di esecuzione, si presume che la funzione funzioni correttamente e restituisca il risultato corretto.

In effetti, stai già praticando questo atto di fede quando usi le funzioni integrate.

Quando chiami "cos" o "exp", non esamini i corpi di quelle funzioni.

Supponi solo che funzionino perché le persone che hanno scritto le funzioni integrate erano buoni programmatori.

Lo stesso vale quando chiami una delle tue funzioni.

Esempio

Ad esempio, in Boolean Functions, abbiamo scritto una funzione chiamata "`isdivisible`" che determina se un numero è divisibile per un altro.

Una volta che ci siamo convinti che questa funzione è corretta, esaminando il codice e testandolo, possiamo usare la funzione senza guardare di nuovo il corpo.

Lo stesso vale per i programmi ricorsivi.

Quando si arriva alla chiamata ricorsiva, invece di seguire il flusso di esecuzione, si dovrebbe presumere che la chiamata ricorsiva funzioni (restituisca il risultato corretto) e poi chiedersi: "Supponendo che io possa trovare il fattoriale di `n - 1`, posso calcolare il fattoriale di `n`?" È chiaro che possiamo, moltiplicando per "`n`".

Certo, è un po 'strano presumere che la funzione funzioni correttamente quando non hai finito di scriverla, ma è per questo che si chiama "atto di fede"!

Section 7

Un altro esempio

Fibonacci function

Dopo “factorial”, l’esempio più comune di una funzione matematica definita ricorsivamente è “fibonacci”, che chiamiamo `fib`:

Fibonacci function

Dopo "factorial", l'esempio più comune di una funzione matematica definita ricorsivamente è "fibonacci", che chiamiamo `fib`:

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

```
function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

Fibonacci function

Dopo "factorial", l'esempio più comune di una funzione matematica definita ricorsivamente è "fibonacci", che chiamiamo `fib`:

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

```
function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

```
fib(n) = n==0 ? 0 : (n==1 ? 1 : fib(n-1) + fib(n-2))
```

Section 8

Ricorsione lineare e non lineare

Benchmarking code

Una grande differenza nel tempo di esecuzione (BenchmarkTools) può essere vista quando si eseguono le funzioni `fact` `efib`:

```
(@v1.5) pkg> add BenchmarkTools

julia> using BenchmarkTools
    [ Info: Precompiling BenchmarkTools [6e4b80f9-dd63-53aa-95

julia> @btime fact(big(40))
    0.011688 ms (242 allocations: 3.98 KiB)
815915283247897734345611269596115894272000000000

julia> @btime fib(40)
    710.000 ms (0 allocations: 0 bytes)
102334155
```

Cerca la differenza

fact(n) = O(n) ; fib(n) = $O(2^n)$

Perché ??

```
function fact(n)
    if n == 0 return 1
    else n * fact(n-1) end
end

function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

Section 9

Controllo dei tipi

Esempio di errore a runtime

Cosa succede se chiamiamo `fact` e diamo `1.5` come argomento?

```
julia> fact(1.5)
ERROR: StackOverflowError:
Stacktrace:
     [1] fact(::Float64) at ./REPL[3]:2
```

Sembra una ricorsione infinita. Come può essere? La funzione ha un caso base: quando "`n == 0`". Ma se "`n`" non è un numero intero, possiamo perdere il caso base e ricorrere per sempre.

Nella prima chiamata ricorsiva, il valore di "`n`" è 0,5. Nel successivo è -0,5. Da lì, diventa più piccolo (più negativo), ma non sarà mai 0.

Abbiamo due scelte. Possiamo provare a generalizzare la funzione fattoriale per lavorare con numeri in virgola mobile, oppure possiamo fare verificare il tipo del suo argomento.

La prima opzione è chiamata funzione `gamma` ed è un po 'oltre lo scopo di questo libro. Quindi optiamo per la seconda.

Controlla i tipi degli argomenti

Possiamo usare l'operatore predefinito `isa` per verificare il `tipo` dell'argomento. Già che ci siamo, possiamo anche assicurarci che l'argomento sia positivo:

```
function fact(n)
    if !(n isa Int64)
        error("Factorial is only defined for integers.")
    elseif n < 0
        error("Factorial is not defined for negative integers.
    elseif n == 0
        return 1
    else
        return n * fact(n-1)
    end
end
```

Istruzioni Guardia

Il primo caso di base gestisce non interi; il secondo gestisce interi negativi.

In entrambi i casi, il programma stampa un messaggio `error` e non restituisce nulla per indicare che qualcosa è andato storto:

```
julia> fact("fred")
ERROR: Factorial is only defined for integers.
julia> fact(-2)
ERROR: Factorial is not defined for negative integers.
```

Se superiamo entrambi i controlli, sappiamo che "`n`" è positivo o zero, quindi possiamo provare che la ricorsione termina.

Questo programma mostra un pattern a volte chiamato guardian. I primi due condizionali fungono da guardie, proteggendo il codice che segue da valori che potrebbero causare un errore. Le guardie consentono di provare la correttezza del codice.

In Catching Exceptions vedremo un'alternativa più flessibile alla stampa di un messaggio di errore: sollevare un'eccezione.

Section 10

Debug

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;
3. C'è qualcosa di sbagliato nel valore restituito o nel modo in cui viene utilizzato; una postcondizione è violata.

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;
3. C'è qualcosa di sbagliato nel valore restituito o nel modo in cui viene utilizzato; una postcondizione è violata.

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;
3. C'è qualcosa di sbagliato nel valore restituito o nel modo in cui viene utilizzato; una postcondizione è violata.

Per escludere la prima possibilità, è possibile aggiungere un'istruzione print all'inizio della funzione e visualizzare i valori dei parametri (e forse i loro tipi).

Oppure si può scrivere codice che controlli esplicitamente le precondizioni.

Se i parametri sembrano buoni, aggiungere un'istruzione `print` prima di ogni istruzione `return` e visualizzare il valore restituito.

Se possibile, controllare manualmente il risultato. Considerare l'idea di chiamare la funzione con valori che semplificano il controllo del risultato (come in Sviluppo incrementale).

Aggiunta di dichiarazioni di stampa

Se la funzione sembra funzionare, controllare la chiamata della funzione per assicurarsi che il valore restituito venga utilizzato correttamente (o che venga utilizzato!).

L'aggiunta di istruzioni `print` all'inizio e alla fine di una funzione può aiutare a rendere più visibile il flusso di esecuzione.

Ad esempio, ecco una versione di `fact` con istruzioni `print`:

```
function fact(n)
    space = " " ^ (4 * n) println(space, "factorial ", n)
    if n == 0
        println(space, "returning 1")
        return 1
    else
        recurse = fact(n-1)
        result = n * recurse
        println(space, "returning ", result) return result
    end
```

Section 11

Glossario

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

sviluppo incrementale Un piano di sviluppo del programma inteso a evitare il debug aggiungendo e testando solo una piccola quantità di codice alla volta.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

sviluppo incrementale Un piano di sviluppo del programma inteso a evitare il debug aggiungendo e testando solo una piccola quantità di codice alla volta.

impalcatura Codice utilizzato durante lo sviluppo del programma ma non fa parte della versione finale.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

sviluppo incrementale Un piano di sviluppo del programma inteso a evitare il debug aggiungendo e testando solo una piccola quantità di codice alla volta.

impalcatura Codice utilizzato durante lo sviluppo del programma ma non fa parte della versione finale.

guardia Un modello di programmazione che utilizza un'istruzione condizionale per verificare e gestire le circostanze che potrebbero causare un errore.

Section 12

Esercizi

aaaa

aaaa

% Fondamenti di Informatica (Elettronici) % THINK JULIA – Capitolo 6 % 3 novembre 2020

6. Funzioni feconde[2]

Molte delle funzioni Julia che abbiamo utilizzato, come le funzioni matematiche, producono valori di ritorno.

Ma le funzioni che abbiamo scritto sono tutte vuote: hanno un effetto, come stampare un valore o spostare una tartaruga, ma restituiscono "`nothing`".

In questo capitolo impareremo a scrivere funzioni fruttuose (o feconde).

Section 13

Valori restituiti

`nothing` come valore di ritorno

La chiamata della funzione genera un valore di ritorno, che di solito assegniamo a una variabile o utilizziamo come parte di un'espressione.

```
e = exp(1.0)
height = radius * sin(radians)
```

Le funzioni che abbiamo scritto finora sono nulle.

Parlando semplicemente, non hanno valore di ritorno; più precisamente, il loro valore di ritorno è `nothing`.

In questo capitolo, scriveremo (finalmente) funzioni feconde. Il primo esempio è `area`, che restituisce l'area di un cerchio con il raggio dato.

Valore "return" implicito

Abbiamo già visto l'istruzione `return` in precedenza, ma in una funzione fruttuosa l'istruzione `return` include un'espressione. Questa istruzione significa: "Torna immediatamente da questa funzione e utilizza la seguente espressione come valore di ritorno."

L'espressione può essere arbitrariamente complicata, quindi avremmo potuto scrivere questa funzione in modo più conciso:

```
function area(radius)
    pi * radius^2
end
```

Il valore restituito da una funzione è il valore dell'ultima espressione valutata, che, per definizione, è l'ultima espressione nel corpo della funzione.

Più istruzioni "return"

D'altra parte, variabili temporanee e dichiarazioni di ritorno esplicite possono semplificare il debug.

A volte è utile avere più istruzioni di ritorno, una in ogni ramo di un condizionale:

```
function absvalue(x)
    if x < 0
        return -x
    else
        return x
    end
end
```

Poiché queste istruzioni `return` sono in un condizionale alternativo, solo una viene eseguita.

Molteplici percorsi di controllo

Non appena viene eseguita un'istruzione `return`, la funzione termina senza eseguire alcuna istruzione successiva.

Il codice che appare dopo un'istruzione `return`, o qualsiasi altro punto in cui il flusso di esecuzione non potrà mai raggiungere, è chiamato codice morto.

In una funzione feconda, è una buona idea assicurarsi che ogni possibile percorso attraverso il programma raggiunga un'istruzione `return`.

```
function absvalue(x)
    if x < 0
        return -x
    end
    if x > 0
        return x
    end
end
```

Valore assoluto

Questa funzione non è corretta perché se `x` è 0, nessuna delle due condizioni è verae la funzione termina senza raggiungere un'istruzione `return`.

Se il flusso di esecuzione arriva alla fine di una funzione, il valore restituito è "`nothing`", che non è il valore assoluto di 0.

```
julia> show(absvalue(0))
nothing
```

Valore assoluto

Questa funzione non è corretta perché se `x` è 0, nessuna delle due condizioni è verae la funzione termina senza raggiungere un'istruzione `return`.

Se il flusso di esecuzione arriva alla fine di una funzione, il valore restituito è "`nothing`", che non è il valore assoluto di 0.

```
julia> show(absvalue(0))
nothing
```

SUGGERIMENTO

Julia fornisce una funzione predefinita chiamata `abs` che calcola i valori assoluti.

Section 14

Sviluppo incrementale

Esempio

Con funzioni più grandi, si può dedicare più tempo al debug

Per gestire programmi sempre più complessi, possiamo utilizzare un processo chiamato sviluppo incrementale.

L'obiettivo dello sviluppo incrementale è evitare lunghe sessioni di debug, aggiungendo e testando solo una piccola quantità di codice alla volta.

Ad esempio, supponiamo di voler trovare la distanza tra due punti, di coordinate $(x1, y1)$ e $(x2, y2)$.

Per il teorema di Pitagora, la distanza è:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Il primo passo è considerare come dovrebbe apparire una funzione di distanza in Julia. In altre parole, quali sono gli input (parametri) e qual è l'output (valore di ritorno)?

In questo caso, gli input sono due punti, che puoi rappresentare utilizzando quattro numeri. Il valore restituito è la distanza, rappresentata da un valore in virgola mobile.

Schema delle funzioni

All'inizio, si può scrivere uno schema della funzione:

```
function distance(x_1, y_1, x_2, y_2)
    0.0
end
```

Ovviamente, questa versione non calcola le distanze; restituisce sempre zero.

Ma è sintatticamente corretta e funziona, il che significa che puoi testarla prima di renderla più complicata. I numeri in pedice sono disponibili nella codifica dei caratteri Unicode (`_1 TAB`, `_2 TAB`, ecc.).

Per testare la nuova funzione, chiamala con argomenti di esempio:

```
distance(1, 2, 4, 6)
```

Ho scelto questi valori in modo che la distanza orizzontale sia 3 e la distanza verticale sia 4; in questo modo, il risultato è 5, l'ipotenusa di un triangolo 3-4-5.

Quando si testa una funzione, è utile conoscere la risposta giusta.

Esempio incrementale

A questo punto abbiamo confermato che la funzione è sintatticamente corretta e possiamo iniziare ad aggiungere codice al corpo.

Un passo successivo ragionevole è calcolare le differenze "`x2 - x1`" e "`y2 - y1`". La prossima versione memorizza questi valori in variabili temporanee e li stampa con la macro @show.

```
function distance(x1, y1, x2, y2)
    dx = x2 - x1
    dy = y2 - y1
    @show dx dy
    0.0
end
```

Se la funzione funziona, dovrebbe visualizzare "`dx = 3`" e "`dy = 4`". Se è così, sappiamo che la funzione sta ottenendo gli argomenti giusti e sta eseguendo correttamente il primo calcolo. In caso contrario, ci sono solo poche righe dacontrollare**.

Versione finale

Successivamente calcoliamo la somma dei quadrati di `dx` e `dy`:

```
function distance(x1, y1, x2, y2)
    dx = x2 - x1
    dy = y2 - y1
    d2 = dx^2 + dy^2
    @show d2
    return sqrt(d2)
end
```

Di nuovo, dovremmo eseguire il programma in questa fase e controllare l'output (che dovrebbe essere 25). Infine, usare `sqrt` per calcolare e restituire il risultato:

Se funziona correttamente, avremo finito. Altrimenti, potremmo voler stampare il valore di `sqrt(d2)` prima dell'istruzione `return`.

Aspetti chiave del processo

La versione finale della funzione non mostra nulla quando viene eseguita; restituisce solo un valore. Le istruzioni di stampa che abbiamo scritto sono utili per il debug, ma una volta che la funzione funziona, occorre rimuoverle.

Un codice del genere è chiamato `scaffolding` perché è utile per la creazione del programma ma non fa parte del prodotto finale.

All'inizio, dovremmo aggiungere solo una o due righe di codice alla volta. Man mano che acquisisci più esperienza, potresti ritrovarti a scrivere e eseguire il debug di blocchi più grandi. In ogni caso, lo sviluppo incrementale può far risparmiare molto tempo per il debug.

Gli aspetti chiave del processo sono:

1. Iniziare con un programma funzionante e apportare piccole modifiche incrementali. In qualsiasi momento, se c'è un errore, si avrà una buona idea di dove si trova.

#. Una volta che il programma sta funzionando, rimuovere* alcune delle impalcature* o consolidare più istruzioni in espressioni composte, ma solo se non rende il programma difficile da leggere

Aspetti chiave del processo

La versione finale della funzione non mostra nulla quando viene eseguita; restituisce solo un valore. Le istruzioni di stampa che abbiamo scritto sono utili per il debug, ma una volta che la funzione funziona, occorre rimuoverle.

Un codice del genere è chiamato `scaffolding` perché è utile per la creazione del programma ma non fa parte del prodotto finale.

All'inizio, dovremmo aggiungere solo una o due righe di codice alla volta. Man mano che acquisisci più esperienza, potresti ritrovarti a scrivere e eseguire il debug di blocchi più grandi. In ogni caso, lo sviluppo incrementale può far risparmiare molto tempo per il debug.

Gli aspetti chiave del processo sono:

1. Iniziare con un programma funzionante e apportare piccole modifiche incrementali. In qualsiasi momento, se c'è un errore, si avrà una buona idea di dove si trova.
2. Utilizzare variabili per contenere valori intermedi in modo da poterli visualizzare e controllare.

#. Una volta che il programma sta funzionando, rimuovere* alcune delle impalcature* o consolidare più istruzioni in espressioni composte, ma solo se non rende il programma difficile da leggere

Section 15

Composizione

Composizione

Come dovremmo aspettarci, possiamo chiamare una funzione dall'interno di un'altra.

Ad esempio, scriveremo una funzione che prende due punti, il centro del cerchio e un punto sul perimetro, e calcola l'area del cerchio.

Supponiamo che il punto centrale sia memorizzato nelle variabili "`xc`" e "`yc`" e che il punto perimetrale sia in "`xp`" e "`yp`".

Il primo passo è trovare il "raggio" del cerchio, che è la distanza tra i due punti. Abbiamo appena scritto una funzione, `distance`, che fa questo:

```
radius = distance(xc, yc, xp, yp)
```

Il passo successivo è trovare l'`area` di un cerchio con quel raggio:

```
result = area(radius)
```

Incapsulamento

"Incapsulando" questi passaggi in una singola funzione, otteniamo:

```
function circlearea(xc, yc, xp, yp)
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
end
```

Le variabili temporanee `radius` e `result` sono utili per lo sviluppo e il debugging, ma una volta che il programma funziona, possiamo renderlo più conciso componendo le chiamate di funzione:

```
function circlearea(xc, yc, xp, yp)
    area(distance(xc, yc, xp, yp))
end
```

Section 16

Funzioni booleane

Funzioni predicato

Le funzioni possono restituire valori booleani, il che è spesso conveniente per nascondere test complicati all'interno delle funzioni. Per esempio:

```
function isdivisible(x, y)
    if x % y == 0
        return true
    else
        return false
    end
end
```

È comune dare nomi di funzioni booleane che suonano come domande sì / no; `isdivisible` restituisce "`true`" o "`false`" per indicare se "`x`" è divisibile per "`y`":

```
julia> isdivisible(6, 4)
false
julia> isdivisible(6, 3)
true
```

Esempi

Il risultato dell'operatore == è un booleano, quindi possiamo scrivere la funzione in modo più conciso restituendola direttamente:

```
function isdivisible(x, y)
    x % y == 0
end
```

Le funzioni booleane vengono spesso utilizzate nelle istruzioni condizionali:

```
if isdivisible(x, y)
    println("x is divisible by y")
end
```

Potremmo essere tentati di scrivere qualcosa come:

```
if isdivisible(x, y) == true
    println("x is divisible by y")
end
```

Ma il confronto extra con "true" non è necessario.

Section 17

Più ricorsione

il sottoinsieme Julia visto finora è Turing-completo

Abbiamo trattato solo un piccolo sottoinsieme di "Julia", ma potresti essere interessato a sapere che questo sottoinsieme è un linguaggio di programmazione completo, il che significa che tutto ciò che può essere calcolato può essere espresso in questo linguaggio.

Qualsiasi programma mai scritto potrebbe essere riscritto utilizzando solo le funzionalità del linguaggio che hai appreso finora (in realtà, avresti bisogno di alcuni comandi per controllare dispositivi come il mouse, i dischi, ecc., ma questo è tutto).

Dimostrare questa affermazione è un esercizio non banale realizzato per la prima volta da Alan Turing, uno dei primi scienziati informatici (alcuni sostengono che fosse un matematico, ma molti dei primi scienziati informatici hanno iniziato come matematici).

Di conseguenza, è conosciuta come la Tesi di Turing.

Per una discussione più completa (e accurata) della Tesi di Turing, raccomando il libro di Michael Sipser Introduzione alla teoria del calcolo.

Per darti un'idea di cosa puoi fare con gli strumenti che hai imparato finora, valuteremo alcune funzioni matematiche definite ricorsivamente.

Una definizione ricorsiva è simile a una definizione circolare, nel senso che la definizione contiene un riferimento alla cosa da definire. Una definizione veramente circolare non è molto utile.

Parole recorsive ;-)

vorpal

Un aggettivo usato per descrivere qualcosa che è vorpal.

Parole recorsive ;-)

vorpal

Un aggettivo usato per descrivere qualcosa che è vorpal.

GNU's Not Unix

Il sistema operativo GNU è un sistema software libero completo, compatibile con Unix. GNU sta per "GNU's Not Unix".

Definizione del fattoriale

Se cerchiamo la definizione della funzione fattoriale, indicata con il simbolo !, potremmo ottenere qualcosa del genere:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n > 0 \end{cases}$$

Questa definizione dice che il fattoriale di 0 è 1 e il fattoriale di qualsiasi altro valore, n, è n moltiplicato per il fattoriale di $n - 1$.

Quindi 3! è 3 per 2!, che è 2 per 1!, che è 1 per 0!. Mettendo tutto insieme, 3! è uguale a 3 per 2 per 1 per 1, che è 6.

vfill pause

Definizione induttiva In matematica e informatica, una definizione ricorsiva o induttiva viene utilizzata per definire gli elementi in un insieme in termini di altri elementi nell'insieme.

Caso base

Se possiamo scrivere una definizione ricorsiva di qualcosa, possiamo scrivere un programma Julia per valutarla. Il primo passo è decidere quali dovrebbero essere i parametri. In questo caso dovrebbe essere chiaro che fattoriale prende un numero intero:

```
function fact(n)
end
```

Se l'argomento è 0, tutto ciò che dobbiamo fare è restituire 1:

```
function fact(n)
    if n == 0
        return 1
    end
end
```

Definizione ricorsiva

Altrimenti, e questa è la parte interessante, dobbiamo fare una chiamata ricorsiva per trovare il fattoriale di `n-1` e poi moltiplicarlo per `n`:

```
function fact(n)
    if n == 0
        return 1
    else
        recurse = fact(n-1)
        result = n * recurse
        return result
    end
end
```

Una riscrittura più compatta e leggibile:

```
function fact(n)
    if n == 0 return 1
    else n * fact(n-1) end
end
```

Flusso di esecuzione

Il flusso di esecuzione di questo programma è simile al flusso del conto alla rovescia in Ricorsione.

Se chiamiamo `fact` con il valore 3:

Since `3` is not `0`, we take the second branch and calculate the factorial of `n-1` ...

Since `2` is not `0`, we take the second branch and calculate the factorial of `n-1` ...

Since `1` is not `0`, we take the second branch and calculate the factorial of `n-1` ...

Since `0` equals `0`, we take the first branch and return `1` without making any more recursive calls.

The return value, `1`, is multiplied by `n`, which is `1`, and the `result` is returned.

The return value, `1`, is multiplied by `n`, which is `2`, and the `result` is returned.

The return value `2` is multiplied by `n`, which is `3`, and the result, `6`, becomes the return value of the function call that started the whole process.

{#

.class width = 100%}

Julia fornisce la funzione "`factorial`" per calcolare il fattoriale di un intero.

Section 18

Atto di fede

Assumendo la correttezza

Seguire il flusso di esecuzione è un modo per leggere i programmi, ma può diventare rapidamente opprimente.

Un'alternativa è quello che io chiamo il "salto della fede".

Quando si arriva a una chiamata di funzione, invece di seguire il flusso di esecuzione, si presume che la funzione funzioni correttamente e restituisca il risultato corretto.

In effetti, stiamo già praticando questo atto di fede quando usiamo le funzioni predefinite.

Quando chiami "cos" o "exp", non esaminiamo i corpi di quelle funzioni.

Supponiamo solo che funzionino perché le persone che hanno scritto le funzioni integrate erano buoni programmatori.

Lo stesso vale quando chiamiamo una delle nostre funzioni.

Esempio

Ad esempio, in Boolean Functions, abbiamo scritto una funzione chiamata "`isdivisible`" che determina se un numero è divisibile per un altro.

Una volta che ci siamo convinti che questa funzione è corretta, esaminando il codice e testandolo, possiamo usare la funzione senza guardare di nuovo il corpo.

Lo stesso vale per i programmi ricorsivi.

Quando si arriva alla chiamata ricorsiva, invece di seguire il flusso di esecuzione, si dovrebbe presumere che la chiamata ricorsiva funzioni (restituisca il risultato corretto) e poi chiedersi: "Supponendo che io possa trovare il fattoriale di `n - 1`, posso calcolare il fattoriale di `n`?" È chiaro che possiamo, moltiplicando per "`n`".

Certo, è un po' strano presumere che la funzione funzioni correttamente quando non hai finito di scriverla, ma è per questo che si chiama "atto di fede"!

Section 19

Un altro esempio

Fibonacci function

Dopo “factorial”, l’esempio più comune di una funzione matematica definita ricorsivamente è “`fibonacci`”, che chiamiamo `fib`:

Fibonacci function

Dopo "factorial", l'esempio più comune di una funzione matematica definita ricorsivamente è "`fibonacci`", che chiamiamo `fib`:

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

```
function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

Fibonacci function

Dopo "factorial", l'esempio più comune di una funzione matematica definita ricorsivamente è "`fibonacci`", che chiamiamo `fib`:

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

```
function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

```
fib(n) = n==0 ? 0 : (n==1 ? 1 : fib(n-1) + fib(n-2))
```

Section 20

Ricorsione lineare e non lineare

Benchmarking code

Una grande differenza nel tempo di esecuzione (BenchmarkTools) può essere vista quando si eseguono le funzioni `fact` `efib`:

```
(@v1.5) pkg> add BenchmarkTools

julia> using BenchmarkTools
    [ Info: Precompiling BenchmarkTools [6e4b80f9-dd63-53aa-95

julia> @btime fact(big(40))
    0.011688 ms (242 allocations: 3.98 KiB)
815915283247897734345611269596115894272000000000

julia> @btime fib(40)
    710.000 ms (0 allocations: 0 bytes)
102334155
```

Cerca la differenza

fact(n) = O(n) ; fib(n) = $O(2^n)$

Perché ??

```
function fact(n)
    if n == 0 return 1
    else n * fact(n-1) end
end

function fib(n)
    if n == 0
        return 0
    elseif n == 1
        return 1
    else
        return fib(n-1) + fib(n-2)
    end
end
```

Section 21

Controllo dei tipi

Esempio di errore a runtime

Cosa succede se chiamiamo `fact` e diamo `1.5` come argomento?

```
julia> fact(1.5)
ERROR: StackOverflowError:
Stacktrace:
     [1] fact(::Float64) at ./REPL[3]:2
```

Sembra una ricorsione infinita. Come può essere? La funzione ha un caso base: quando "`n == 0`". Ma se "`n`" non è un numero intero, possiamo perdere il caso base e ricorrere per sempre.

Nella prima chiamata ricorsiva, il valore di "`n`" è 0,5. Nel successivo è -0,5. Da lì, diventa più piccolo (più negativo), ma non sarà mai 0.

Abbiamo due scelte. Possiamo provare a generalizzare la funzione fattoriale per lavorare con numeri in virgola mobile, oppure possiamo verificare il tipo del suo argomento.

La prima opzione è chiamata funzione `gamma` ed è un po 'oltre lo scopo di questo libro. Quindi optiamo per la seconda.

Controlla i tipi degli argomenti

Possiamo usare l'operatore predefinito `isa` per verificare il `tipo` dell'argomento. Già che ci siamo, possiamo anche assicurarci che l'argomento sia positivo:

```
function fact(n)
    if !(n isa Int64)
        error("Factorial is only defined for integers.")
    elseif n < 0
        error("Factorial is not defined for negative integers.
    elseif n == 0
        return 1
    else
        return n * fact(n-1)
    end
end
```

Istruzioni Guardia

Il primo caso di base gestisce non interi; il secondo gestisce interi negativi.

In entrambi i casi, il programma stampa un messaggio `error` e non restituisce nulla per indicare che qualcosa è andato storto:

```
julia> fact("fred")
ERROR: Factorial is only defined for integers.
julia> fact(-2)
ERROR: Factorial is not defined for negative integers.
```

Se superiamo entrambi i controlli, sappiamo che "`n`" è positivo o zero, quindi possiamo provare che la ricorsione termina.

Questo programma mostra un pattern a volte chiamato guardian. I primi due condizionali fungono da guardie, proteggendo il codice che segue da valori che potrebbero causare un errore. Le guardie consentono di provare la correttezza del codice.

In Catching Exceptions vedremo un'alternativa più flessibile alla stampa di un messaggio di errore: sollevare un'eccezione.

Section 22

Debug

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;
3. C'è qualcosa di sbagliato nel valore restituito o nel modo in cui viene utilizzato; una postcondizione è violata.

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;
3. C'è qualcosa di sbagliato nel valore restituito o nel modo in cui viene utilizzato; una postcondizione è violata.

Checkpoint naturali per il debug

La suddivisione di un programma di grandi dimensioni in funzioni più piccole crea punti di controllo naturali per il debug.

Se una funzione non funziona, ci sono tre possibilità da considerare:

1. C'è qualcosa di sbagliato negli argomenti che la funzione sta ottenendo; una precondizione è violata.
2. C'è qualcosa di sbagliato nella funzione;
3. C'è qualcosa di sbagliato nel valore restituito o nel modo in cui viene utilizzato; una postcondizione è violata.

Per escludere la prima possibilità, è possibile aggiungere un'istruzione print all'inizio della funzione e visualizzare i valori dei parametri (e forse i loro tipi).

Oppure si può scrivere codice che controlli esplicitamente le precondizioni.

Se i parametri sembrano buoni, aggiungere un'istruzione `print` prima di ogni istruzione `return` e visualizzare il valore restituito.

Se possibile, controllare manualmente il risultato. Considerare l'idea di chiamare la funzione con valori che semplificano il controllo del risultato (come in Sviluppo incrementale).

Aggiunta di dichiarazioni di stampa

Se la funzione sembra funzionare, controllare la chiamata della funzione per assicurarsi che il valore restituito venga utilizzato correttamente (o che venga utilizzato!).

L'aggiunta di istruzioni `print` all'inizio e alla fine di una funzione può aiutare a rendere più visibile il flusso di esecuzione.

Ad esempio, ecco una versione di `fact` con istruzioni `print`:

```
function fact(n)
    space = " " ^ (4 * n) println(space, "factorial ", n)
    if n == 0
        println(space, "returning 1")
        return 1
    else
        recurse = fact(n-1)
        result = n * recurse
        println(space, "returning ", result) return result
    end
```

Section 23

Glossario

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

sviluppo incrementale Un piano di sviluppo del programma inteso a evitare il debug aggiungendo e testando solo una piccola quantità di codice alla volta.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

sviluppo incrementale Un piano di sviluppo del programma inteso a evitare il debug aggiungendo e testando solo una piccola quantità di codice alla volta.

impalcatura Codice utilizzato durante lo sviluppo del programma ma non fa parte della versione finale.

Glossario

variabile temporanea Una variabile utilizzata per memorizzare un valore intermedio in un calcolo complesso.

codice morto Parte di un programma che non può mai essere eseguita, spesso perché appare dopo un'istruzione `return`.

sviluppo incrementale Un piano di sviluppo del programma inteso a evitare il debug aggiungendo e testando solo una piccola quantità di codice alla volta.

impalcatura Codice utilizzato durante lo sviluppo del programma ma non fa parte della versione finale.

guardia Un modello di programmazione che utilizza un'istruzione condizionale per verificare e gestire le circostanze che potrebbero causare un errore.

Section 24

Esercizi

aaaa

aaaa