Fondamenti di Informatica (Elettronici)

THINK JULIA – Capitolo 16

4 dicembre 2020

16. Strutture e funzioni[1]

1. Tempo
2. Funzioni pure
3. Modificatori
4. Prototipazione contro pianificazione
5. Debug
6. Glossario
7. Esercizi

[1] Tratto da https://benlauwens.github.io/ThinkJulia.jl/latest/book.html, disponibile sotto Licenza 'Creative Commons Attribution-NonCommercial 3.0 Unported'.

Ora che sappiamo come creare nuovi tipi compositi, il passaggio successivo consiste nello scrivere funzioni che accettano oggetti definiti dal programmatore come parametri e li restituiscono come risultati.

In questo capitolo presentiamo anche lo "stile di programmazione funzionale" e due nuove strategie di sviluppo del programma.

Section 1

Tempo

MyTime: esempio di struct

Come altro esempio di un tipo composto, definiremo una `struct` chiamata `MyTime` che registra l'ora del giorno.

La definizione `struct` ha questo aspetto:

```
"""
Represents the time of day.
fields: hour, minute, second
"""
struct MyTime
    hour
    minute
    second
end
```

Il nome "`Time`" è già utilizzato in Julia e per evitare un conflitto di nomi, scegliamo "`MyTime`". Possiamo creare un nuovo oggetto `MyTime`:

```
julia> time = MyTime(11, 59, 30)
MyTime(11, 59, 30)
```

Diagramma dell'oggetto

Il diagramma oggetto per l'oggetto `MyTime` è simile al diagramma dell'oggetto in figura

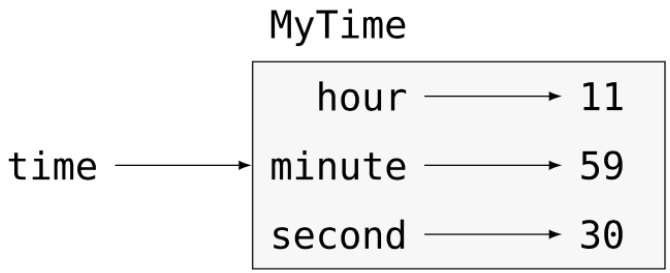

Figure 22. Object diagram

Esercizio 16-1

Scrivete una funzione chiamata `printtime` che prenda un oggetto `MyTime` e lo stampi nella forma ora: minuto: secondo. La macro `@printf` del modulo `Printf` di `StdLib` stampa un numero intero con la sequenza di formato "%02d" utilizzando almeno due cifre, incluso uno zero iniziale se necessario.

Esercizio 16-2

Scrivete una funzione booleana chiamata `isafter` che accetta due oggetti `MyTime`, `t1` e `t2`, e restituisce `true` se `t1` segue cronologicamente `t2` e false altrimenti. Sfida: non utilizzare un'istruzione `if`.

Section 2

Funzioni pure

Strategia di progetto: Prototipo e patch

Nelle prossime sezioni scriveremo due funzioni che sommano valori temporali.

Mostrano due tipi di funzioni: funzioni pure e modificatori.

Dimostrano anche un piano di sviluppo che chiamerò prototipo e patch, che è un modo per affrontare un problema complesso partendo da un semplice prototipo e affrontando in modo incrementale le complicazioni.

Ecco un semplice prototipo di "`addtime`":

```
function addtime(t1, t2)
    MyTime(t1.hour + t2.hour, t1.minute + t2.minute, t1.second + t2
end
```

Prototipo addtime: inizio del lavoro

La funzione crea un nuovo oggetto `MyTime`, inizializza i suoi campi e restituisce un riferimento al nuovo oggetto.

Questa è chiamata una funzione pura perché non modifica nessuno degli oggetti passati come argomenti e non ha alcun effetto, come visualizzare un valore o ottenere l'input dell'utente, oltre alla restituzione di un valore.

Per testare questa funzione, creerò due oggetti `MyTime`: `start` contiene l'ora di inizio di un film, come Monty Python e Holy Grail, e `duration` contiene la durata del film, che è un'ora e 35 minuti.

"`addtime`" scopre quando il film sarà finito.

```
julia> start = MyTime(9, 45, 0);
julia> duration = MyTime(1, 35, 0);
julia> done = addtime(start, duration);
julia> printtime(done)
10:80:00
```

Raffinamento del prototipo "addtime"

Il risultato, 10:80:00, potrebbe non essere quello che volevamo. Il problema è che questa funzione non si occupa dei casi in cui il numero di secondi o minuti supera i sessanta.

Quando ciò accade, dobbiamo "portare" i "secondi" extra nella colonna "minuti" o i "minuti" extra nella colonna "ora". Ecco una versione migliorata:

```
function addtime(t1, t2)
    second = t1.second + t2.second
    minute = t1.minute + t2.minute
    hour = t1.hour + t2.hour
    if second >= 60
        second -= 60
        minute += 1
    end
    if minute >= 60
        minute -= 60
        hour += 1
    end
    MyTime(hour, minute, second)
end
```

Sebbene questa funzione sia corretta, sta iniziando a diventare lunga. Più avanti vedremo un'alternativa più breve.

Section 3

Modificatori

Funzione di modifica "`increment!`"

A volte è utile che una funzione modifichi gli oggetti che ottiene come parametri.

In tal caso, le modifiche sono visibili al chiamante. Le funzioni che funzionano in questo modo sono chiamate modificatori.

`increment!`, che aggiunge un dato numero di secondi a un oggetto `MyTime`, può essere scritta naturalmente come un modificatore. Ecco una bozza approssimativa:

```
function increment!(time, seconds)
    time.second += seconds
    if time.second >= 60
        time.second -= 60
        time.minute += 1
    end
    if time.minute >= 60
        time.minute -= 60
        time.hour += 1
    end
end
```

Ulteriori affinamenti

La prima riga esegue l'operazione di base; il resto riguarda i casi speciali che abbiamo visto prima.

Questa funzione è corretta? Cosa succede se i secondi sono molto maggiori di 60?

In tal caso, non è sufficiente "riportare" una volta; dobbiamo continuare a farlo fino a quando "time.second" è inferiore a sessanta.

Una soluzione è sostituire le istruzioni "`if`" con le istruzioni "`while`".

Ciò renderebbe la funzione corretta, ma non molto efficiente.

Ulteriori affinamenti

La prima riga esegue l'operazione di base; il resto riguarda i casi speciali che abbiamo visto prima.

Questa funzione è corretta? Cosa succede se i secondi sono molto maggiori di 60?

In tal caso, non è sufficiente "riportare" una volta; dobbiamo continuare a farlo fino a quando "time.second" è inferiore a sessanta.

Una soluzione è sostituire le istruzioni "`if`" con le istruzioni "`while`".

Ciò renderebbe la funzione corretta, ma non molto efficiente.

Esercizio 16-3

Scrivi una versione corretta di `increment!` che non contenga alcun loop.

Stile di programmazione funzionale

Tutto ciò che può essere fatto con i modificatori può essere fatto anche con funzioni pure.

In effetti, alcuni linguaggi di programmazione consentono solo funzioni pure.

Esistono alcune prove che i programmi che utilizzano funzioni pure sono più veloci da sviluppare e meno soggetti a errori rispetto ai programmi che utilizzano modificatori.

Ma i modificatori a volte sono convenienti e i programmi funzionali tendono ad essere meno efficienti.

In generale, ti consiglio di scrivere funzioni pure ogni volta che è ragionevole e di ricorrere a modificatori solo se c'è un vantaggio convincente.

Questo approccio potrebbe essere definito uno stile di programmazione funzionale.

Stile di programmazione funzionale

Tutto ciò che può essere fatto con i modificatori può essere fatto anche con funzioni pure.

In effetti, alcuni linguaggi di programmazione consentono solo funzioni pure.

Esistono alcune prove che i programmi che utilizzano funzioni pure sono più veloci da sviluppare e meno soggetti a errori rispetto ai programmi che utilizzano modificatori.

Ma i modificatori a volte sono convenienti e i programmi funzionali tendono ad essere meno efficienti.

In generale, ti consiglio di scrivere funzioni pure ogni volta che è ragionevole e di ricorrere a modificatori solo se c'è un vantaggio convincente.

Questo approccio potrebbe essere definito uno stile di programmazione funzionale.

Esercizio 16-4

Scrivi una versione "pura" di `increment` che crei e restituisca un nuovo oggetto `MyTime` invece di modificare il parametro.

Section 4

Prototipazione contro pianificazione

Modello (pattern) di sviluppo progettato

Il piano di sviluppo che sto dimostrando si chiama "prototipo e patch".

- Per ogni funzione, ho scritto un prototipo che ha eseguito il calcolo di base e poi l'ho testato, correggendo gli errori lungo il percorso.

Modello (pattern) di sviluppo progettato

Il piano di sviluppo che sto dimostrando si chiama "prototipo e patch".

- Per ogni funzione, ho scritto un prototipo che ha eseguito il calcolo di base e poi l'ho testato, correggendo gli errori lungo il percorso.
- Questo approccio può essere efficace, soprattutto se non hai ancora una conoscenza approfondita del problema.

Modello (pattern) di sviluppo progettato

Il piano di sviluppo che sto dimostrando si chiama "prototipo e patch".

- Per ogni funzione, ho scritto un prototipo che ha eseguito il calcolo di base e poi l'ho testato, correggendo gli errori lungo il percorso.
- Questo approccio può essere efficace, soprattutto se non hai ancora una conoscenza approfondita del problema.
- Ma le correzioni incrementali possono generare codice inutilmente complicato, poiché si occupa di molti casi speciali, e inaffidabile, poiché è difficile sapere se sono stati rilevati tutti gli errori.

Modello (pattern) di sviluppo progettato

Il piano di sviluppo che sto dimostrando si chiama "prototipo e patch".

- Per ogni funzione, ho scritto un prototipo che ha eseguito il calcolo di base e poi l'ho testato, correggendo gli errori lungo il percorso.
- Questo approccio può essere efficace, soprattutto se non hai ancora una conoscenza approfondita del problema.
- Ma le correzioni incrementali possono generare codice inutilmente complicato, poiché si occupa di molti casi speciali, e inaffidabile, poiché è difficile sapere se sono stati rilevati tutti gli errori.
- Un'alternativa è lo sviluppo progettato, in cui una visione di alto livello del problema può rendere la programmazione molto più semplice.

Modello (pattern) di sviluppo progettato

Il piano di sviluppo che sto dimostrando si chiama "prototipo e patch".

- Per ogni funzione, ho scritto un prototipo che ha eseguito il calcolo di base e poi l'ho testato, correggendo gli errori lungo il percorso.
- Questo approccio può essere efficace, soprattutto se non hai ancora una conoscenza approfondita del problema.
- Ma le correzioni incrementali possono generare codice inutilmente complicato, poiché si occupa di molti casi speciali, e inaffidabile, poiché è difficile sapere se sono stati rilevati tutti gli errori.
- Un'alternativa è lo sviluppo progettato, in cui una visione di alto livello del problema può rendere la programmazione molto più semplice.
- In questo caso, l'intuizione è che un oggetto `Time` è in realtà un numero di tre cifre in base 60 (vedere https: //en.wikipedia.org/wiki/Sexagesimal)!

Modello (pattern) di sviluppo progettato

Il piano di sviluppo che sto dimostrando si chiama "prototipo e patch".

- Per ogni funzione, ho scritto un prototipo che ha eseguito il calcolo di base e poi l'ho testato, correggendo gli errori lungo il percorso.
- Questo approccio può essere efficace, soprattutto se non hai ancora una conoscenza approfondita del problema.
- Ma le correzioni incrementali possono generare codice inutilmente complicato, poiché si occupa di molti casi speciali, e inaffidabile, poiché è difficile sapere se sono stati rilevati tutti gli errori.
- Un'alternativa è lo sviluppo progettato, in cui una visione di alto livello del problema può rendere la programmazione molto più semplice.
- In questo caso, l'intuizione è che un oggetto `Time` è in realtà un numero di tre cifre in base 60 (vedere https: //en.wikipedia.org/wiki/Sexagesimal)!
- Il secondo attributo è la "colonna delle unità", l'attributo dei minuti è la "colonna dei sessanta" e l'attribute dell'ora è la "colonna delle trentasei centinaia".

Funzione timetoint

Quando abbiamo scritto addtime e increment!, stavamo effettivamente facendo l'addizione in base 60, motivo per cui dovevamo passare da una colonna all'altra.

Questa osservazione suggerisce un altro approccio all'intero problema: possiamo convertire gli oggetti "MyTime" in numeri interi e trarre vantaggio dal fatto che il computer sa come eseguire operazioni aritmetiche su interi.

Ecco una funzione che converte "mytimes" in numeri interi:

```
function timetoint(time)
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
end
```

Versione più corta dell'originale e più facile da verificare

Ed ecco una funzione che converte un intero in un `MyTime` (ricorda che `divrem` divide il primo argomento per il secondo e restituisce il quoziente e il resto come una tupla):

```
function inttotime(seconds)
    (minutes, second) = divrem(seconds, 60)
    hour, minute = divrem(minutes, 60)
    MyTime(hour, minute, second)
end
```

Versione più corta dell'originale e più facile da verificare

Ed ecco una funzione che converte un intero in un `MyTime` (ricorda che `divrem` divide il primo argomento per il secondo e restituisce il quoziente e il resto come una tupla):

```
function inttotime(seconds)
    (minutes, second) = divrem(seconds, 60)
    hour, minute = divrem(minutes, 60)
    MyTime(hour, minute, second)
end
```

Potrebbe essere necessario pensare un po' ed eseguire alcuni test per convincersi che queste funzioni sono corrette.

Un modo per testarle è controllare che `timetoint(inttotime(x)) == x` per molti valori di `x`. Questo è un esempio di controllo di coerenza.

Una volta che sei convinto che siano corrette, puoi usarle per riscrivere `addtime`:

```
function addtime(t1, t2)
    seconds = timetoint(t1) + timetoint(t2)
    inttotime(seconds)
end
```

Riscrivi "`increment!`" usando "`timetoint`" e "`inttotime`"

In un certo senso, la conversione da base 60 a base 10 e viceversa è più difficile che gestire i tempi.

La conversione di base è più astratta; la nostra intuizione nell'affrontare i valori del tempo è migliore.

Ma se abbiamo l'intuizione di trattare i tempi come numeri in base 60 e fare l'investimento nella scrittura delle** funzioni di conversione** (`timetoint` `einttotime`), otteniamo un programma più breve, più facile da leggere ed eseguire il debug e più affidabile.

Riscrivi "`increment!`" usando "`timetoint`" e "`inttotime`"

In un certo senso, la conversione da base 60 a base 10 e viceversa è più difficile che gestire i tempi.

La conversione di base è più astratta; la nostra intuizione nell'affrontare i valori del tempo è migliore.

Ma se abbiamo l'intuizione di trattare i tempi come numeri in base 60 e fare l'investimento nella scrittura delle** funzioni di conversione** (`timetoint einttotime`), otteniamo un programma più breve, più facile da leggere ed eseguire il debug e più affidabile.

- È anche più facile aggiungere funzionalità in un secondo momento.

Riscrivi "`increment!`" usando "`timetoint`" e "`inttotime`"

In un certo senso, la conversione da base 60 a base 10 e viceversa è più difficile che gestire i tempi.

La conversione di base è più astratta; la nostra intuizione nell'affrontare i valori del tempo è migliore.

Ma se abbiamo l'intuizione di trattare i tempi come numeri in base 60 e fare l'investimento nella scrittura delle** funzioni di conversione** (`timetoint einttotime`), otteniamo un programma più breve, più facile da leggere ed eseguire il debug e più affidabile.

- È anche più facile aggiungere funzionalità in un secondo momento.
- Immagina di sottrarre due `MyTime` per trovare l'intervallo tra di loro.

Riscrivi "`increment!`" usando "`timetoint`" e "`inttotime`"

In un certo senso, la conversione da base 60 a base 10 e viceversa è più difficile che gestire i tempi.

La conversione di base è più astratta; la nostra intuizione nell'affrontare i valori del tempo è migliore.

Ma se abbiamo l'intuizione di trattare i tempi come numeri in base 60 e fare l'investimento nella scrittura delle** funzioni di conversione** (`timetoint` `einttotime`), otteniamo un programma più breve, più facile da leggere ed eseguire il debug e più affidabile.

- È anche più facile aggiungere funzionalità in un secondo momento.
- Immagina di sottrarre due `MyTime` per trovare l'intervallo tra di loro.
- L'approccio ingenuo sarebbe quello di implementare la sottrazione con il prestito.

Riscrivi "`increment!`" usando "`timetoint`" e "`inttotime`"

In un certo senso, la conversione da base 60 a base 10 e viceversa è più difficile che gestire i tempi.

La conversione di base è più astratta; la nostra intuizione nell'affrontare i valori del tempo è migliore.

Ma se abbiamo l'intuizione di trattare i tempi come numeri in base 60 e fare l'investimento nella scrittura delle** funzioni di conversione** (`timetoint` `einttotime`), otteniamo un programma più breve, più facile da leggere ed eseguire il debug e più affidabile.

- È anche più facile aggiungere funzionalità in un secondo momento.
- Immagina di sottrarre due `MyTime` per trovare l'intervallo tra di loro.
- L'approccio ingenuo sarebbe quello di implementare la sottrazione con il prestito.
- L'utilizzo delle funzioni di conversione sarebbe più semplice e più probabile che sia corretto.

Riscrivi "increment!" usando "timetoint" e "inttotime"

In un certo senso, la conversione da base 60 a base 10 e viceversa è più difficile che gestire i tempi.

La conversione di base è più astratta; la nostra intuizione nell'affrontare i valori del tempo è migliore.

Ma se abbiamo l'intuizione di trattare i tempi come numeri in base 60 e fare l'investimento nella scrittura delle** funzioni di conversione** (`timetoint einttotime`), otteniamo un programma più breve, più facile da leggere ed eseguire il debug e più affidabile.

- È anche più facile aggiungere funzionalità in un secondo momento.
- Immagina di sottrarre due `MyTime` per trovare l'intervallo tra di loro.
- L'approccio ingenuo sarebbe quello di implementare la sottrazione con il prestito.
- L'utilizzo delle funzioni di conversione sarebbe più semplice e più probabile che sia corretto.
- Ironia della sorte, a volte rendere un problema più difficile (o più generale) lo

Section 5

Debug

Invariant predicate requirements

Un oggetto "`MyTime`" è ben formato se i valori di "`minute`" e "`second`" sono compresi tra 0 e 60 (incluso 0 ma non 60) e se "`hour`" è positivo.

"`hour`" e "`minute`" dovrebbero essere valori interi, ma potremmo consentire a "`second`" di avere una parte frazionaria.

Requisiti come questi sono chiamati invarianti perché dovrebbero sempre essere `veri`.

In altre parole, se le invarianti "sono false", qualcosa è andato storto.

Scrivi il codice per controllare le invarianti

Scrivere codice per controllare le invarianti può aiutare a rilevare gli errori e trovarne le cause.

Ad esempio, potresti avere una funzione come `isvalidtime` che accetta un oggetto `MyTime` e restituisce `false` se viola un invariante:

```
function isvalidtime(time)
    if time.hour < 0 || time.minute < 0 || time.second < 0
        return false
    end
    if time.minute >= 60 || time.second >= 60
        return false
    end
    true
end
```

Verificare la validità degli argomenti

All'inizio di ogni funzione puoi controllare gli argomenti per assicurarti che siano validi:

```
function addtime(t1, t2)
    if !isvalidtime(t1) || !isvalidtime(t2)
        error("invalid MyTime object in add_time")
    end
    seconds = timetoint(t1) + timetoint(t2)
    inttotime(seconds)
end
```

@assert macro

Oppure potresti usare una macro `@assert`, che controlla un dato invariante e genera un'eccezione se fallisce:

```
function addtime(t1, t2)
    @assert(isvalidtime(t1) && isvalidtime(t2), "invalid MyTim
    seconds = timetoint(t1) + timetoint(t2)
    inttotime(seconds)
end
```

Le macro `@assert` sono utili perché distinguono il codice che si occupa di condizioni normali dal codice che controlla gli errori.

Section 6

Glossario

Glossario

prototipo e patch Un piano di sviluppo che prevede la scrittura di una bozza approssimativa di un programma, il test e la correzione degli errori non appena vengono rilevati.

Glossario

prototipo e patch Un piano di sviluppo che prevede la scrittura di una bozza approssimativa di un programma, il test e la correzione degli errori non appena vengono rilevati.

sviluppo progettato Un piano di sviluppo che implica una visione di alto livello del problema e una pianificazione maggiore rispetto allo sviluppo incrementale o allo sviluppo di prototipi.

Glossario

prototipo e patch Un piano di sviluppo che prevede la scrittura di una bozza approssimativa di un programma, il test e la correzione degli errori non appena vengono rilevati.

sviluppo progettato Un piano di sviluppo che implica una visione di alto livello del problema e una pianificazione maggiore rispetto allo sviluppo incrementale o allo sviluppo di prototipi.

funzione pura Una funzione che non modifica nessuno degli oggetti che riceve come argomenti. La maggior parte delle funzioni pure sono fruttuose.

Glossario

prototipo e patch Un piano di sviluppo che prevede la scrittura di una bozza approssimativa di un programma, il test e la correzione degli errori non appena vengono rilevati.

sviluppo progettato Un piano di sviluppo che implica una visione di alto livello del problema e una pianificazione maggiore rispetto allo sviluppo incrementale o allo sviluppo di prototipi.

funzione pura Una funzione che non modifica nessuno degli oggetti che riceve come argomenti. La maggior parte delle funzioni pure sono fruttuose.

modificatore Una funzione che modifica uno o più oggetti ricevuti come argomenti. La maggior parte dei modificatori sono nulli; cioè, non restituiscono nulla.

Glossario

prototipo e patch Un piano di sviluppo che prevede la scrittura di una bozza approssimativa di un programma, il test e la correzione degli errori non appena vengono rilevati.

sviluppo progettato Un piano di sviluppo che implica una visione di alto livello del problema e una pianificazione maggiore rispetto allo sviluppo incrementale o allo sviluppo di prototipi.

funzione pura Una funzione che non modifica nessuno degli oggetti che riceve come argomenti. La maggior parte delle funzioni pure sono fruttuose.

modificatore Una funzione che modifica uno o più oggetti ricevuti come argomenti. La maggior parte dei modificatori sono nulli; cioè, non restituiscono nulla.

stile di programmazione funzionale Uno stile di progettazione del programma in cui la maggior parte delle funzioni è pura.

Glossario

prototipo e patch Un piano di sviluppo che prevede la scrittura di una bozza approssimativa di un programma, il test e la correzione degli errori non appena vengono rilevati.

sviluppo progettato Un piano di sviluppo che implica una visione di alto livello del problema e una pianificazione maggiore rispetto allo sviluppo incrementale o allo sviluppo di prototipi.

funzione pura Una funzione che non modifica nessuno degli oggetti che riceve come argomenti. La maggior parte delle funzioni pure sono fruttuose.

modificatore Una funzione che modifica uno o più oggetti ricevuti come argomenti. La maggior parte dei modificatori sono nulli; cioè, non restituiscono nulla.

stile di programmazione funzionale Uno stile di progettazione del programma in cui la maggior parte delle funzioni è pura.

invariante Una condizione logica che non dovrebbe mai cambiare durante l'esecuzione di un programma.

Section 7

Esercizi

aaaa

aaaa