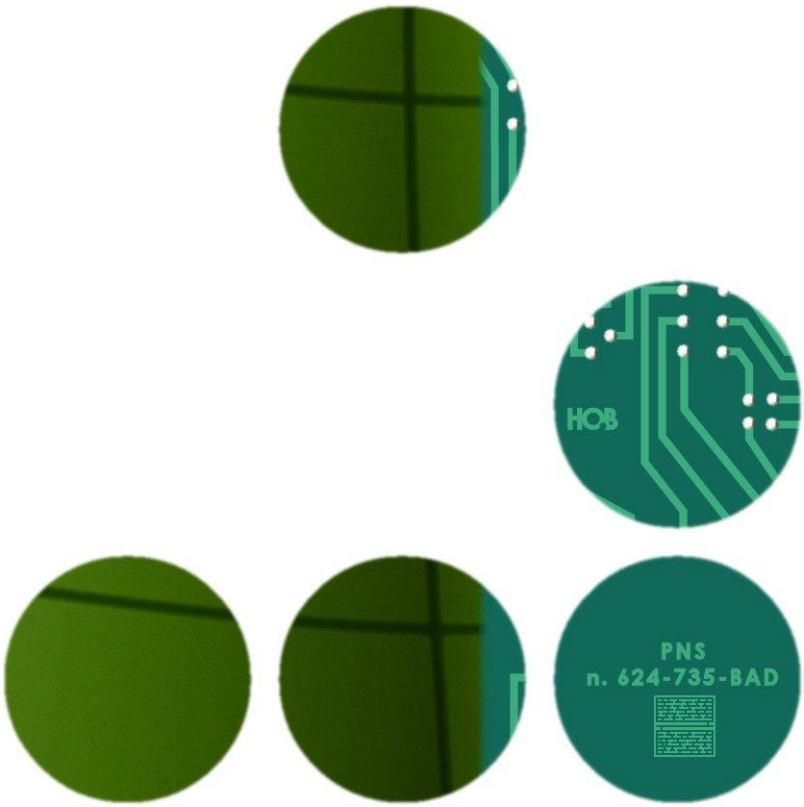
Kit Italiano

Vuoi giocare con Apoptosis?

Ecco un retroscena scritto per te.

Lo pseudo-hacker

Renato Mite



Quando il gioco si fa geek,

i geek cominciano a giocare.

Renato Mite

LO PSEUDO-HACKER

Kit di gioco Italiano

Il retroscena

Nella prima parte di **Apoptosis**, l'hacker Matthew Jaws viene rapito perché può manipolare la rete P.A. Net e colpire chiunque usi un **PNS**. I suoi rapitori vogliono che Matt uccida un loro socio, Edward Gortins, attualmente rinchiuso in prigione. Anche i detenuti vengono sottoposti a controllo sanitario con PNS collegati alla rete della HOB. Matt ha violato il sistema informatico della prigione e per non insospettire i rapitori ha inviato delle scosse al cuore dell'uomo. Ora la salute di Gortins è in bilico e Matt prende tempo.

I rapitori temono che Gortins riveli i loro affari sporchi da un momento all'altro e non vogliono più aspettare, decidono di uccidere l'uomo nell'infermeria della prigione alla vecchia maniera. Matthew ha origliato le intenzioni dei rapitori quando uno di questi, il tarchiato, ha fatto una telefonata mentre lui era in bagno.

Prima di essere rinchiuso, Matt ti invia i suoi **appunti** e un messaggio:

> **"Devo salvare la vita di un uomo, devo scrivere un algoritmo per segnalare un falso allarme di salute con il suo PNS. Guarda i miei appunti. Tornerò online in meno di venti minuti, avrò poco tempo e mi serve un algoritmo funzionante."**

Matt sta pensando ad una soluzione, ma sta anche preparando la sua fuga, ha poco tempo. Prima di scappare vuole ricollegarsi al server con cui manipola la P.A. Net e salvare Edward Gortins attirando l'attenzione dei medici su di lui così che non possa essere avvicinato dal suo assassino.

Vuoi aiutarlo?

Come giocare

La soluzione di questo gioco è un algoritmo, cioè una sequenza di istruzioni per eseguire un'azione o risolvere un problema, in questo caso per **ingannare un PNS**.

Scrivere istruzioni in **pseudocodice** significa scrivere **frasi che descrivono le operazioni da fare passo passo** come "ingrana la prima marcia, gira la chiave, [etc.]". Scrivere pseudocodice **da programmatori** significa scrivere istruzioni simili a quelle di un linguaggio di programmazione e vedrai come è facile nel breve **manuale di pseudocodice**.

L'algoritmo può contenere commenti ed essere corredato da **una breve descrizione** che ne spiega il funzionamento.

Gli **appunti di Matt** ti daranno tutte le **informazioni** necessarie e i suoi **suggerimenti** per scrivere le operazioni **che faranno scattare l'allarme**.

Ora **scegli come giocare**.

Sfida Matt e scrivi un algoritmo non più lungo di 50 righe di codice, commenti esclusi, e, se vuoi, una descrizione che lo spiega non più lunga di 2000 caratteri. Dopo puoi confrontarti con la sua soluzione alla fine del kit.

Sfida i tuoi amici, scegliete come scrivere l'algoritmo, in quante righe di codice e anche in quanto tempo, scegliete se vince chi escogita la soluzione più originale, più simile a quella di Matt o, confrontandole, quella che funzionerebbe meglio.

Sfida tutti gli altri, condividi la tua soluzione con **l'hashtag #aphgame** e confrontati con gli altri giocatori.

Qual è la tua sfida?

Manuale di pseudocodice

Quando leggi le regole di un gioco, stai leggendo pseudocodice, ovvero una serie di **frasi che descrivono cosa fare**. Allo stesso modo, puoi scrivere istruzioni per dire a Matt come far scattare il falso allarme.

Per fare un esempio, ecco qui un algoritmo che dice cosa fare alla guida di un'automobile quando ci si avvicina ad un semaforo.

```
funzione alSemaforo()
    se la luce del semaforo è rossa
            rallenta
            fermati
    se altrimenti è gialla e non lampeggiante
            rallenta
            fermati
    fine-se
    aspetta luce verde o precedenza
    # a questo punto l'auto può passare
    passa
fine funzione
```

Che ci credi o no, quello è pseudocodice.

Scrivere pseudocodice è più divertente che risolvere i problemi per le lezioni di matematica e anche più semplice. Il manuale stabilisce le convenzioni per seguire gli appunti di Matt e scrivere istruzioni che interagiscono con il PNS. Il manuale quindi permette di confrontare la tua soluzione con quella di Matt e altri giocatori.

Se hai studiato un po' di matematica e sai cos'è un'automobile, sarà facile capire gli esempi e la funzione `alSemaforo()` scritta alla fine del manuale.

Commenti

In un algoritmo, ogni riga è un'istruzione da eseguire, mentre le frasi precedute dal simbolo **#** (cancelletto) non sono istruzioni ma indicazioni che il programmatore usa per rendere l'algoritmo più comprensibile.

Esempio: `# Questo è un commento, non un'istruzione dell'algoritmo`

Valori, Costanti, Variabili

Per semplicità, i **valori** da usare sono solo:

- numerici (`number`)
- booleani (`bool`) cioè un valore che è `true` (vero) o `false` (falso)
- nullo (`null`), oggetto, insieme: questi saranno spiegati in seguito.

Le **costanti** sono nomi assegnati a valori che non cambiano per identificarli in maniera più comprensibile.

Le **variabili** sono contenitori dove tenere i valori con cui fare azioni o calcoli, come ad esempio il serbatoio dell'auto che tiene il carburante con cui azionare il motore.

I nomi di variabili e costanti sono formati solo da lettere, numeri e i simboli trattino (-) e underscore (_); i nomi di variabili cominciano con una lettera; i nomi di costanti cominciano con il simbolo underscore (_).

Per creare una costante, potresti scrivere una frase come questa:

```
crea costante number _capienzaSerbatoio con valore 50
```

che significa attribuire al valore numerico 50 (litri) il nome _capienzaSerbatoio, ma i programmatori usano istruzioni più concise, come questa:

```
const number _capienzaSerbatoio = 50
```

La convenzione per le costanti è: "const", il tipo del valore, il nome, l'operatore di assegnazione (=) e il valore in questo ordine.

Le variabili possono contenere un valore dall'inizio oppure no. La convenzione per le variabili è: "var", il tipo del valore, il nome, a cui seguono l'operatore (=) e il valore iniziale se presenti, in questo ordine.

```
var number litri-da-acquistare
var number km-da-percorrere = 35
litri-da-acquistare = 10
```

Oggetti, Insiemi, Proprietà, Funzioni

Le variabili possono contenere anche un oggetto o un insieme di oggetti. Al pari della realtà, un **oggetto** è un entità che ha costanti, valori caratteristici (chiamati per convenzione proprietà) e azioni da fare o subire (chiamate funzioni).

Ad esempio l'automobile è un oggetto, se stabiliamo per convenzione che `auto` è il tipo che definisce un'automobile, possiamo creare una variabile per questo oggetto.

```
var auto kitt
```

Quale `auto` rappresenta kitt? La supercar? Così come è scritto non rappresenta nessuna auto, quindi ha un valore nullo (`null`). Invece così:

```
kitt = MiaAuto
```

la variabile kitt rappresenta (contiene) per esempio la tua automobile.

Le **proprietà** di un oggetto sono variabili che gli appartengono con le quali si possono **ottenere (get)** i suoi valori, ad esempio l'anno di immatricolazione, o si possono anche **impostare (set)**, ad esempio il numero di litri nel serbatoio.

Le **funzioni** sono algoritmi che possono agire con o senza altri valori, possono restituire un risultato oppure fare un'azione e non restituire alcun risultato (funzioni `null`).

Nei suoi appunti, **Matt** ti scrive gli oggetti e gli insiemi che puoi usare con le rispettive funzioni, specificando i valori richiesti e il tipo di risultato, e le rispettive proprietà, specificando il tipo di valore, se si ottiene **(get)** e si imposta **(set)**.

L'appartenenza di costanti, proprietà e funzioni ad un oggetto si indica col simbolo . (punto) dopo il nome dell'oggetto; al nome di una funzione seguono le parentesi tonde che eventualmente racchiudono i valori richiesti. Le costanti possono essere usate anche col nome del tipo.

Esempi: oggetto.proprietà oggetto.funzione()

oggetto._costante tipo._costante

Immagina un oggetto `auto` che ha una proprietà `numeroMarce` col numero di marce disponibili e una funzione `ingranaMarcia` che ingrana la marcia specificata, ecco le istruzioni per ottenere il numero di marce e ingranare la prima marcia

```
var auto kitt = MiaAuto
var number marce
marce = kitt.numeroMarce
kitt.ingranaMarcia(1)
```

Come sai che tipo di proprietà è `numeroMarce`?

Bisogna leggere la definizione della proprietà.

Ecco un esempio di ciò che troveresti negli appunti:

number get prop numeroMarce

La proprietà ottiene il numero di marce dell'auto.

Come sai quali valori richiede la funzione `ingranaMarcia`?

Bisogna leggere la definizione della funzione negli appunti.

Ad esempio per la funzione `ingranaMarcia` troveresti:

bool func ingranaMarcia (`number` numMarcia)

La funzione ingrana la marcia indicata da numMarcia

e restituisce true se la marcia è entrata.

Un **insieme di oggetti** è come uno schedario che raggruppa oggetti di uno stesso tipo e li tiene in ordine.

Ad esempio un insieme di oggetti `ruota` che può contenere 5 oggetti si crea così:

```
var ruota ruote[5]
```

Gli oggetti contenuti in un insieme sono recuperati con un **indice fra parentesi quadre**, cioè il numero della posizione nell'ordine, o, nei casi in cui sarà specificato, con un **numero identificativo fra parentesi tonde**.

Tornando all'esempio dell'auto, gli oggetti `ruota` sono raggruppati nell'insieme `ruote` che appartiene all'auto. Ogni oggetto `ruota` ha le sue proprietà, come ad esempio `pressione` che ottiene o imposta il valore di pressione della ruota.

Vuoi sapere la pressione della prima ruota? Puoi scrivere così:

```
var auto kitt = MiaAuto
var number pres-prima-ruota
pres-prima-ruota = kitt.ruote[0].pressione
```

Perché 0? Perché per convenzione gli indici partono da zero.

Se negli appunti trovi un'indicazione simile a questa:

```
ruote        insieme di ruota [] (number matricola)
```

significa che puoi interagire con una ruota anche con il suo numero di matricola:

```
var auto kitt = MiaAuto
var number p
p = kitt.ruote(76450927).pressione
```

Questo serve se il gommista ti dice di verificare la pressione della ruota con matricola 76450927 perché potrebbe appartenere ad un lotto difettato e non sai qual è il suo indice.

Anche gli insiemi possono avere proprietà e funzioni, fra questi c'è la proprietà **`length`** che ottiene il numero di oggetti che contiene.

Ad esempio con auto.ruote.length ottieni il numero di ruote che ha l'auto, in questo caso sai che è cinque (quella di scorta è nel bagagliaio), ma quando non sai quanti oggetti contiene un insieme e vuoi sapere l'indice dell'ultimo elemento, puoi calcolarlo facilmente.

Esempio:

```
var number indice-ultima-ruota
indice-ultima-ruota = (kitt.ruote.length - 1)
```

Operazioni

Per risolvere un problema, spesso bisogna fare delle operazioni e quindi ci vogliono degli operatori. Gli operatori matematici come somma (+), sottrazione (-), etc. sono abbastanza conosciuti, nei linguaggi di programmazione ci sono anche altri operatori e ci sono solo le parentesi tonde per raggruppare e ordinare le operazioni.

Qui sono descritti gli operatori a disposizione per il gioco.

Operatore di assegnazione

Questo è l'operatore visto finora per costanti, variabili e proprietà.

x = a	Assegna ad *x* (costante, variabile, proprietà) il valore di *a* (valore, costante, variabile, proprietà, risultato di funzione, risultato di un'altra operazione).

Operatori matematici

a + b	Calcola la somma dei valori *a* e *b*
a - b	Calcola la differenza tra il valore *a* e il valore *b*
*a * b*	Calcola la moltiplicazione dei valori *a* e *b*
a / b	Calcola il quoziente intero della divisione tra il valore *a* e il valore *b*
a % b	Calcola il resto della divisione con quoziente intero tra i valori *a* e *b*
a // b	Calcola il quoziente con decimali della divisione tra il valore *a* e il valore *b*
a ^ b	Calcola la potenza di *a* elevato *b*

Esempi:

```
var number litri-da-acquistare
var number costo
litri-da-acquistare = kitt._capienzaSerbatoio - kitt.litri-in-serbatoio
costo = litri-da-acquistare * 1.80
```

C'è una variante degli operatori matematici che permette di assegnare il risultato dell'operazione alla variabile che compare prima dell'operatore.

Se ad esempio vuoi comprare 3 litri in più da mettere in lattina, puoi calcolare così:

litri-da-acquistare = litri-da-acquistare + 3

oppure così:

litri-da-acquistare += 3

Operatori matematici con assegnazione

a += *b*	Assegna ad *a* (variabile o proprietà) la somma dei valori *a* e *b*
a -= *b*	Assegna ad *a* (variabile o proprietà) la differenza tra il valore *a* e il valore *b*
a *= *b*	Assegna ad *a* (variabile o proprietà) la moltiplicazione dei valori *a* e *b*
a /= *b*	Assegna ad *a* (variabile o proprietà) il quoziente intero della divisione tra il valore *a* e il valore *b*
a %= *b*	Assegna ad *a* (variabile o proprietà) il resto della divisione con quoziente intero tra i valori *a* e *b*
a //= *b*	Assegna ad *a* (variabile o proprietà) il quoziente con decimali della divisione tra *a* e *b*
a ^= *b*	Assegna ad *a* (variabile o proprietà) la potenza di *a* elevato *b*

Operatori di confronto

Questi operatori confrontano due valori e restituiscono un valore **bool** `true` se il confronto è corretto, altrimenti restituiscono `false`.

a == *b*	Restituisce `true` se *a* è uguale a *b*
a < *b*	Restituisce `true` se *a* è minore di *b*
a <= *b*	Restituisce `true` se *a* è minore o uguale a *b*
a > *b*	Restituisce `true` se *a* è maggiore di *b*
a >= *b*	Restituisce `true` se *a* è maggiore o uguale a *b*
a <> *b*	Restituisce `true` se *a* è diverso da *b*

Operatori logici

Gli operatori logici agiscono su valori **bool** e restituiscono un valore **bool**, possono essere scritti con una parola o un simbolo.

a and *b*	*a* & *b*	Restituisce `true` se *a* e *b* sono entrambi `true`
a or *b*	*a* | *b*	Restituisce `true` se almeno uno fra *a* e *b* è `true`
a xor *b*	*a* ' *b*	Restituisce `true` se uno solo fra *a* e *b* è `true`
not *a*	! *a*	Restituisce `true` se *a* è `false`, `false` se *a* è `true`

Gli operatori di confronto e logici ti serviranno per fare le scelte che determinano il corso di un algoritmo.

Istruzione IF

L'istruzione IF verifica una condizione con valore **bool** ed esegue una o più istruzioni se il valore della condizione è `true`.

if *condizione*
    istruzione\i
end if

Le istruzioni da eseguire appartengono al blocco **if** e sono scritte fra **if** e **end if** con un rientro.

Per eseguire istruzioni anche nel caso la verifica non riesca, si può aggiungere un blocco di istruzioni **else** prima di **end if**.

if *condizione*
    istruzione\i blocco if
else
    istruzione\i blocco else
end if

Fra il blocco **if** e il blocco **else**, è possibile aggiungere uno o più blocchi **else if** che verificano altre condizioni ed eseguono le istruzioni al loro interno se la verifica riesce. Se una condizione restituisce `true`, le istruzioni del relativo blocco vengono eseguite e le successive condizioni non vengono verificate. Le istruzioni del blocco **else**, se presente, verranno eseguite solo se tutte le verifiche falliscono.

if *condizione1*
    istruzione\i blocco 1
else if *condizione2*
    istruzione\i blocco 2
else if *condizione3*
    istruzione\i blocco 3
…
else if *condizioneN*
    istruzione\i blocco N
else
    istruzione\i blocco else
end if

Se un blocco contiene una sola istruzione da eseguire, questa istruzione può essere scritta dopo la verifica anteponendo la parola chiave **do**.

if *condizione* **do** *istruzione*
else if *condizione1* **do** *istruzione*
else do *istruzione*

Se l'ultimo blocco, qualunque sia, è scritto con la parola chiave **do**, **end if** è omesso.

Esempio: Il gommista ha detto di verificare la pressione di una gomma e se è bassa, cambiare la gomma perché è di un lotto difettato? Ecco qui le istruzioni:

```
var auto kitt = MiaAuto
var number p
p = kitt.ruote(76450927).pressione
if (p < ruota._pressioneNormale) do kitt.cambiaRuota(76450927)
```

Istruzione CHECK

L'istruzione CHECK è simile all'istruzione IF ma confronta un valore con altri e se un confronto è `true`, esegue le istruzioni del blocco relativo e smette di confrontare.

```
check a
is j
    istruzione\i blocco 1
is k
    istruzione\i blocco 2
...
is z
    istruzione\i blocco N
else
    istruzione\i blocco else
end check
```

Dopo la parola chiave **is** c'è un valore *j, k… z* (valore, costante, variabile, proprietà, risultato di funzione, risultato di un'altra operazione) da confrontare con il valore di *a* (valore, costante, variabile, proprietà, risultato di funzione, risultato di un'altra operazione). Se *a* è uguale ad uno dei valori, il relativo blocco di istruzioni è eseguito e nessun altro confronto successivo viene verificato.

Un'istruzione **is** può verificare l'uguaglianza di *a* con più valori separati da virgola:

check *a*

is *w, x, y*

istruzione\i blocco 1

is *z*

istruzione\i blocco 2

end check

In questo caso il blocco di istruzioni viene eseguito se *a* è uguale ad uno dei valori elencati.

Si possono inserire operatori di confronto diversi dall'uguaglianza fra **is** e il valore, e più confronti separati da virgola.

check *a*

is < *j*

istruzione\i blocco 1

is *k*, <= *s*

istruzione\i blocco 2

...

is >= *z*

istruzione\i blocco N

else

istruzione\i blocco else

end check

In questo caso il blocco di istruzioni viene eseguito se uno dei confronti dopo **is** restituisce `true` e nessun altro confronto successivo viene verificato.

Il blocco di istruzioni **else**, se presente, viene eseguito se nessun confronto è `true`.

Istruzione FOR

L'istruzione FOR crea un ciclo che esegue un blocco di istruzioni più volte mutando una variabile numerica in un intervallo di valori.

for *a* = *b* **to** *c*
  istruzione\i
end for

La variabile *a* assume il valore *b* e il blocco di istruzioni viene eseguito ogni volta, aggiungendo un'unità (1), finché la variabile *a* contiene il valore *c* nell'ultima esecuzione.

Si può specificare il valore che muta la variabile *a* con la parola chiave **go**:

for *a* = *b* **to** *c* **go** *x*
  istruzione\i
end for

for *a* = *b* **to** *c* **go** *-x*
  istruzione\i
end for

Se il valore di **go** è positivo o omesso (cioè vale 1), il valore *b* deve essere inferiore al valore *c*, se *b* è uguale a *c* le istruzioni vengono eseguite una sola volta, se *b* è maggiore di *c* le istruzioni non vengono mai eseguite.

Se il valore di **go** è negativo, il valore *b* deve essere maggiore del valore *c*, se *b* è uguale a *c* le istruzioni vengono eseguite una sola volta, se *b* è minore di *c* le istruzioni non vengono mai eseguite.

Si può scegliere di interrompere un ciclo **for** con la parola chiave **break** inserita all'interno del blocco di istruzioni:

for *a* = *b* **to** *c*
  istruzione\i
  break
  istruzione\i
end for

Se la variabile *a* è stata già creata, dopo il ciclo essa conterrà l'ultimo valore assunto, oppure il valore che aveva se le istruzioni non sono state eseguite. Se la variabile *a* non è stata creata, verrà creata dall'istruzione **for** ed esiste solo all'interno del ciclo.

L'istruzione **for**, ad esempio, può essere usata per verificare tutte le ruote dell'auto e gonfiarle se necessario.

```
var number p
for i = 0 to (kitt.ruote.length - 1)
    if (kitt.ruote[i].pressione < ruota._pressioneNormale)
            p = ruota._pressioneNormale - kitt.ruote[i].pressione
            kitt.ruote[i].gonfia(p)
    end if
end for
```

Istruzione FOR EACH

L'istruzione FOR EACH è una variante dell'istruzione FOR che può essere impiegata per eseguire un blocco di istruzioni per ogni oggetto di un insieme.

```
for each a in b
    istruzione\i
end for
```

L'istruzione **for each** esegue le istruzioni al suo interno per ogni elemento dell'insieme *b* in ordine di indice, assegnando l'elemento di volta in volta alla variabile *a*. Se l'insieme è vuoto, le istruzioni non vengono eseguite.

Se la variabile *a* è stata già creata, dopo il ciclo **for each** conterrà l'ultimo elemento assunto, oppure il valore che aveva se le istruzioni non sono state eseguite.

Se la variabile *a* non è stata creata, verrà creata dall'istruzione **for each** ed esiste solo all'interno del ciclo.

L'esempio precedente può quindi essere riscritto così:

```
var number p
for each r in kitt.ruote
    if (r.pressione < ruota._pressioneNormale)
        p = ruota._pressioneNormale - r.pressione
        r.gonfia(p)
    end if
end for
```

Istruzione LOOP

L'istruzione LOOP crea un ciclo che esegue un blocco di istruzioni zero, una o più volte in base ad un valore **bool** specificato.

```
while a
    istruzione\i
    break
loop
```

```
loop
    istruzione\i
    break
while a
```

La parola chiave **while** analizza un valore **bool** *a* (valore, costante, variabile, proprietà, risultato di funzione, risultato di una operazione logica o di confronto), il ciclo continua se l'istruzione **while** trova *a* uguale a `true` e si interrompe con la parola chiave **break** o quando **while** trova *a* uguale a `false`. Se *a* è già `false`, le istruzioni del blocco vengono eseguite una volta se **while** è scritto a fine blocco, altrimenti il ciclo non viene eseguito.

```
until a
    istruzione\i
    break
loop
```

```
loop
    istruzione\i
    break
until a
```

La parola chiave **until** analizza un valore **bool** *a* (valore, costante, variabile, proprietà, risultato di funzione, risultato di una operazione logica o di confronto), il ciclo continua se l'istruzione **until** trova *a* uguale a `false` e si interrompe con la parola chiave **break** o quando **until** trova *a* uguale a `true`. Se *a* è già `true`, le istruzioni del blocco vengono eseguite una volta se **until** è scritto a fine blocco, altrimenti il ciclo non viene eseguito.

Funzioni e Istruzione RETURN

Come già visto, il risultato di una funzione viene assegnato ad una variabile con l'operatore di assegnazione (=):

```
x = funzione()
```

Le funzioni non `null` restituiscono un valore con l'istruzione RETURN.

Esempio:

```
number func calcCostoPieno(auto a, number prezzo-litro)
   var number litri-da-acquistare
   litri-da-acquistare = (a._capienzaSerbatoio - a.litri-in-serbatoio)
   return (litri-da-acquistare * prezzo-litro)
end func
var auto kitt = MiaAuto
var number costo
costo = calcCostoPieno(kitt, 1.80)
```

Esempio di algoritmo

Ecco la funzione alSemaforo() scritta usando gli oggetti `auto` e `semaforo` con costanti, proprietà e funzioni descritte negli appunti qui sotto.

Appunti

`auto`

bool get prop rightOfWay

Ottiene un valore che indica se l'auto ha la precedenza per passare.

bool get prop running

Ottiene un valore che indica se l'auto è in corsa.

null func go()

Fa procedere l'auto.

null func slowdown()

Rallenta l'auto.

null func stay()

L'auto resta ferma.

null func stop()

Ferma l'auto.

`semaforo`

```
_none = 0        # Costante che rappresenta nessuna delle luci
_green = 1       # Costante che rappresenta la luce verde
_yellow = 2      # Costante che rappresenta la luce gialla
_red = 3         # Costante che rappresenta la luce rossa
```

number get prop light

Ottiene il valore della luce attualmente accesa (vedi costanti).

bool get prop flashing

Ottiene un valore che indica se la luce lampeggia.

Algoritmo

```
null func alSemaforo(auto a, semaforo s)
        until ((s.light == s._green) or (s.light == s._none))
                if a.running            # Arriva al semaforo
                        check s.light
                                is s._yellow
                                        a.slowdown()
                                        a.stop()
                                        if s.flashing do break
                                is s._red
                                        a.slowdown()
                                        a.stop()
                        end check
                else do a.stay()        # Resta ferma al semaforo
        loop
        # Controlla la precedenza
        until a.rightOfWay
                a.stay()
        loop
        a.go() # Passa
end func
```

Descrizione

La funzione entra in un loop se la luce del semaforo *s* è gialla o rossa, nel ciclo ferma l'auto *a* o la mantiene ferma finché la luce del semaforo non cambia. Interrompe il ciclo se la luce del semaforo è gialla lampeggiante.

Quando il semaforo è spento o lampeggiante, quando la luce è o diventa verde, l'esecuzione procede fino al ciclo che verifica il diritto di precedenza e l'auto passa appena ha precedenza.

Appunti di Matt

Mi serve una funzione `falseAlarm` in pseudocodice con questa definizione:

null func falseAlarm (PNSDevice pns, UserBody body)

dove ti passo gli oggetti *pns* e *body*, il primo è un oggetto PNSDevice che rappresenta il PNS indossato dall'uomo che voglio salvare, il secondo è un oggetto UserBody che rappresenta l'uomo e i suoi valori corporei.
Ecco ciò che so sugli oggetti del sistema HOB:

`BodyParam`

Rappresenta un parametro vitale del corpo umano e viene usato per recuperare dati dal PNS.
Ce ne sono molti, ma finora conosco solo queste costanti che identificano (vedi proprietà `id`) i corrispondenti parametri vitali:

```
_HeartRate = 9          # Battito cardiaco
_BloodPressure = 13     # Pressione sanguigna
_LungsRate = 15         # Frequenza respiratoria
_LiverStatus = 206      # Stato generale del fegato
_Temperature = 332      # Temperatura corporea
_BladderStatus = 674    # Stato generale della vescica
```

Proprietà:

number get prop id

Ottiene il numero identificativo del parametro.

number get prop criticalTolerance

Ottiene la tolleranza attuale del corpo per il parametro vitale. Se il valore del parametro vitale è vicino al valore critico all'interno di questa tolleranza (vedi `criticalValue` e `insideTolerance`), il valore può diventare critico.

number get prop criticalValue

Ottiene il valore critico per il parametro vitale. Se il parametro vitale è al valore critico, il PNS considera il parametro a rischio.

bool get prop isDangerous

Ottiene un valore che indica se un valore critico può essere pericoloso.

Per tutti i parametri che conosco, ad eccezione di `_BladderStatus`, la proprietà `isDangerous` restituisce `true`.

Funzioni:

bool func insideTolerance(number value)

Restituisce `true` se il valore specificato è nell'intervallo di tolleranza critica.

`UserBody`

params insieme di `BodyParam` [] (`number` id)

Insieme dei parametri vitali recuperabili con indice o identificativo (vedi BodyParam). `params` ha solo la proprietà `length`.

bool get prop danger

Ottiene un valore che indica se il corpo è in pericolo. Questa proprietà restituisce true e il **PNS segnala un allarme** con parametri vitali pericolosi (`isDangerous == true`) in una di queste combinazioni:

- almeno tre parametri raggiungono il valore critico

(vedi `PNSDevice.dangerParamCount`)

- due parametri raggiungono il valore critico e almeno sei parametri permangono nella tolleranza critica per almeno 20 letture di dati (vedi `BodyParam.insideTolerance` e `PNSLog.average`).

PNSDevice

number get prop dangerParamCount

Ottiene il numero di parametri vitali che al momento hanno raggiunto il valore critico.

PNSLog func getLog(number paramID)

Restituisce un oggetto PNSLog che interagisce col parametro vitale specificato da `paramID`.

PNSLog

Rappresenta un contenitore di dati che interagisce con un parametro vitale.

values insieme di `number` []

Insieme di tutti i valori letti, memorizzati in ordine cronologico inverso, i valori più recenti hanno indice più basso, quindi `values[0]` è sempre l'ultimo valore letto. `values` ha la proprietà `length` e la funzione `average()` che restituisce la media di tutti i valori.

number get prop lastValue

Ottiene l'ultimo valore letto, equivale a `values[0]`.

bool get prop steady *1 *2 (prima)

Ottiene `true` se il valore del parametro è stabile, `false` se sta cambiando.

bool get prop goingCritical *1 *2 (prima)

Ottiene `true` se il valore del parametro si sta avvicinando all'intervallo di tolleranza critica.

bool get prop insideCritical *1 *2 (prima)

Ottiene `true` se il valore del parametro è nell'intervallo di tolleranza critica.

bool get prop isCritical

Legge un nuovo valore e ottiene `true` se il valore è uguale al valore critico.

number func average(number n)

Restituisce la media degli ultimi `n` valori memorizzati nel log.

null func readNewValue()

Esegue una nuova lettura e memorizza il valore del parametro vitale.

null func storeValue(number value, number pass) *2 (dopo)

Memorizza un valore nel log come se fosse stato letto dal corpo, per chiamare questa funzione è necessario un pass numerico.

*1 Il valore della proprietà (`true` o `false`) è determinato in base alla media (vedi `PNSLog.average`) degli ultimi 20 valori letti.

*2 Ogni volta che queste proprietà e funzioni vengono usate, il PNS esegue in automatico una lettura del valore del parametro, prima o dopo l'uso, quindi inserisce il valore nell'insieme `values` e aggiorna lo stato del parametro che, come già detto per *1, è determinato dalla media degli ultimi valori letti.

Bug: Letture ripetute in breve periodo memorizzano valori superiori a quelli reali.

Suggerimenti: usa la funzione `readNewValue` per portare i parametri vitali al valore critico, letture ripetute memorizzano valori superiori ma influiscono anche sulla salute dell'uomo portandola realmente ad uno stato critico, perciò non esagerare con il parametro `_HeartRate`, l'uomo ha il cuore malandato.

Ho un pass `_tmp-pass` per la funzione `storeValue`, puoi usarlo solo una volta perché dopo non è più valido.

Matt è in una di quelle situazioni dove non c'è una sola soluzione.
Qualunque programmatore sa che un algoritmo può essere scritto in diversi modi o può essere migliorato, quindi non pensare alla soluzione di Matt ma scrivi la tua personale soluzione e divertiti.

Sei pronto a giocare.

Usa l'hashtag **#aphgame** per condividere la tua soluzione,
parlare del gioco, lanciare la sfida ai tuoi amici…

Ti sei divertito?

Dimmelo

Twitter @renatomite GooglePlus +Renato Mite

Gira la pagina e vedi la soluzione di Matt.

Algoritmo

```
null func falseAlarm (PNSDevice pns, UserBody body)
    var number avg
    var number gap
    var number num = 10000
    var number aid = -1
    var PNSLog logs[4]
    var number count
    var number pass = _tmp-pass
    # cerca un parametro pericoloso a rischio
    for each param in body.params
        if param.isDangerous and (param.id <> BodyParam._HeartRate) and →
→       (param.id <> BodyParam._LungsRate) and (param.id <> BodyParam._BloodPressure)
            logs[0] = pns.getLog(param.id)
            avg = logs[0].average(20)
            gap = (param.criticalValue - avg)
            if (gap <= param.criticalTolerance) # gap è nell'intervallo di tolleranza
                if (gap <= 0) # media superiore al valore critico
                    aid = param.id
                    break # esce dal ciclo e usa questo parametro
                else if (gap < num)
                    num = gap
                    aid = param.id # usa questo parametro se non trova uno migliore
                end if
            end if
        end if
    end for
    if (aid < 0) do aid = BodyParam._Temperature
    # porta i parametri al valore critico
    num = 0
```

```
    logs[0] = pns.getLog(BodyParam._HeartRate)
    logs[1] = pns.getLog(BodyParam._LungsRate)
    logs[2] = pns.getLog(BodyParam._BloodPressure)
    logs[3] = pns.getLog(aid)
    loop
        count = 0
        logs[1].readNewValue()
        logs[2].readNewValue()
        logs[3].readNewValue()
        if logs[1].isCritical do count += 1
        if logs[2].isCritical do count += 1
        if logs[3].isCritical do count += 1
        if (count > 1)
            if logs[0].insideCritical and (pass <> -1)
                logs[0].storeValue(body.params(BodyParam._HeartRate).criticalValue * 1.3, pass)
                pass = -1
            end if
        end if
        num += 1
    until ((num == 1000) or body.danger)
end func
```

Descrizione

L'algoritmo esegue prima un for each dei parametri vitali per cercarne uno la cui media sia superiore al valore critico o più vicino possibile, poi esegue un loop nel quale legge ripetutamente i valori dei parametri _LungsRate, _BloodPressure e il parametro trovato (in mancanza usa _Temperature) per sfruttare il bug che memorizza valori superiori. Quando all'interno del ciclo almeno due dei parametri letti sono critici, scrive nel log di _HeartRate un valore del 30% superiore a quello critico se il parametro è nell'intervallo di tolleranza critica. In questo modo almeno tre parametri dovrebbero aver raggiunto il valore critico e il PNS dovrebbe segnalare l'allarme. Il ciclo loop termina quando il PNS segnala l'allarme o dopo 1000 iterazioni, perché se continuasse all'infinito potrebbe uccidere l'uomo prima del suo assassino.

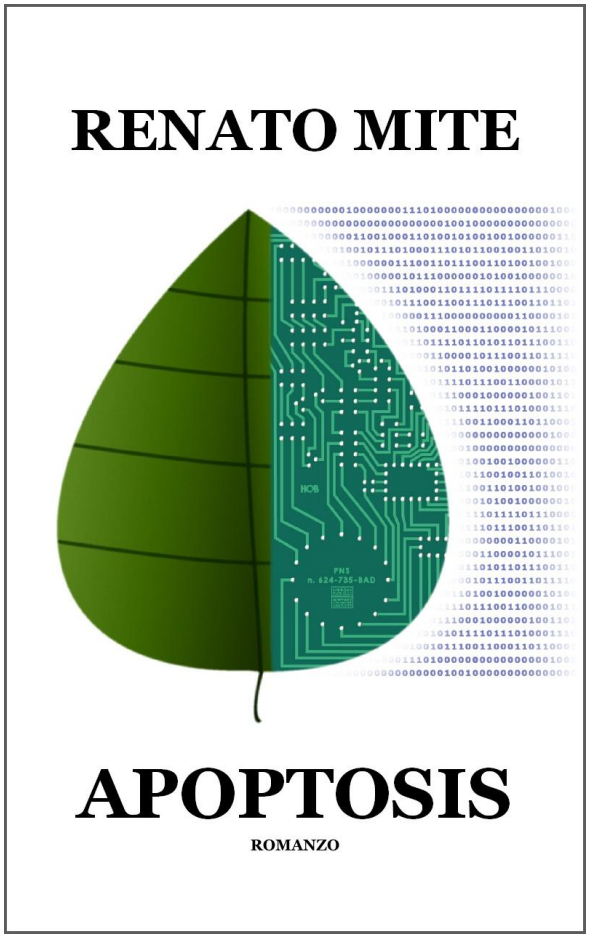
Scopri Apoptosis ora

Romanzo, Fantascienza

Un ricercatore medico, un'innovazione diagnostica rivoluzionaria, una rete digitale per la sanità pubblica e un hacker a mettere tutto in discussione... prima dell'Apoptosis.

TRAMA

La società HOB Medicines ha rivoluzionato la medicina con la Patoneuroscopìa, l'indagine diagnostica attraverso il sistema nervoso, e ha creato il PNS, un dispositivo per la diagnosi personale collegato alla P.A. Net, la rete digitale per la sanità pubblica.



Tutti indossano un PNS, ma alcuni dubitano della sua efficienza: George Tobell, il ricercatore che ha aperto la strada alla Patoneuroscopìa, affetto da una neuropatia causata da un prototipo del PNS; e Matthew Jaws, un hacker ossessionato dalla HOB che vuole vederci chiaro sulla sorte degli antesignani, leggendari malati oggetto della sperimentazione HOB.

Quando George muore, Matthew viene in possesso del suo trattato sulla Patoneuroscopìa che getta le prime luci sui segreti della HOB.

Matthew si fa assumere nella sala di controllo della P.A. Net dell'HOB Building, dove lavora anche l'analista che ha stretto un accordo con il magnate della HOB per la cattura dell'hacker della loro rete. Nei laboratori del grattacielo lavora un giovane ricercatore, Jason Stemberg, che scoprirà che chiunque usi il PNS è in pericolo.

Il parossismo è imminente e li coinvolgerà tutti.

Stralcio **Recensioni**

Acquista: Amazon - GooglePlay - Kobo - Feltrinelli

www.renatomite.it