Little Man Computer

PROGETTO DEL CORSO DI LINGUAGGI DI PROGRAMMAZIONE
ANNO ACCADEMICO 2018–2019. APPELLO DI GENNAIO 2019.
CONSEGNA DOMENICA 13 GENNAIO 2019, ORE 23:55

Marco Antoniotti, Luca Manzoni, Gabriella Pasi

OVERVIEW DEL PROGETTO

Il *little man computer* (LMC) è un semplice modello di computer creato per scopi didattici. Esso possiede 100 celle di memoria (numerate da 0 a 99) ognuna della quali può contenere un numero da 0 a 999 (estremi inclusi). Il computer possiede un solo registro, detto *accumulatore*, una coda di input ed una coda di output. LMC possiede un numero limitato di tipi di istruzioni ed un equivalente assembly altrettanto semplificato. Lo scopo del progetto è quello di produrre

A. Un simulatore del LMC che dato il contenuto iniziale delle memoria (una lista di 100 numeri) e una sequenza di valori di input simuli il comportamento del LMC e produca il contenuto della coda di output dopo l'arresto del LMC.
B. Un assembler che, dato un file scritto nell'assembly semplificato del LMC produca il contenuto iniziale della memoria.

La parte rimanente di questo testo dettaglierà l'architettura del LMC, la sintassi dell'assembly e i predicati/funzioni che le implementazioni in Prolog e Common Lisp devono implementare.

ARCHITETTURA DEL LMC

Il LMC è composto dalle seguenti componenti:

A. Una **memoria** di 100 celle numerate tra 0 e 99. Ogni cella può contenere un numero tra 0 e 999. Non viene effettuata alcuna distinzione tra dati e istruzioni: a seconda del momento il contenuto di una certa cella può essere interpretato come un'istruzione (con una semantica che verrà spiegata in seguito) oppure come un dato (e, ad esempio, essere sommato ad un altro valore).
B. Un **registro accumulatore** (inizialmente zero).
C. Un **program counter**. Il program counter contiene l'indirizzo dell'istruzione da eseguire ed è inizialmente zero. Se *non* viene sovrascritto da altre istruzioni (salti condizionali e non condizionali) viene incrementato di uno ogni volta che un'istruzione viene eseguita. Se raggiunge il valore 999 e viene incrementato il valore successivo è zero.
D. Una **coda di input**. Questa coda contiene tutti i valori forniti in input al LMC, che sono necessariamente numeri compresi tra 0 e 999. Ogni volta che l'LMC legge un valore di input esso viene eliminato dalla coda.
E. Una **coda di output**. Questa coda è inizialmente vuota e contiene tutti i valori mandati in output dal LMC, che sono necessariamente numeri compresi tra 0 e 999. La coda è strutturata in modo tale da avere in testa il primo output prodotto e come ultimo elemento l'ultimo output prodotto: i valori di output sono quindi in ordine cronologico.
F. Un **flag**. Ovvero un singolo bit che può essere acceso o spento. Inizialmente esso è zero (flag assente). Solo le istruzioni aritmetiche modificano il valore del flag e, in particolare, un flag a uno (flag presente) indica che l'ultima operazione aritmetica eseguita ha prodotto un risultato maggiore di 999 o minore di 0. Un flag assente indica invece che il valore prodotto

dall’ultima operazione aritmetica ha prodotto un risultato compreso tra 0 e 999.

La seguente tabella rappresenta come le istruzioni presenti in memoria debbano essere interpretate:

Valore	Nome Istruzione	Significato
1xx	Addizione	Somma il contenuto della cella di memoria xx con il valore contenuto nell'accumulatore e scrive il valore risultante nell'accumulatore. Il valore salvato nell'accumulatore è la somma modulo 1000. Se la somma non supera 1000 il flag è impostato ad assente, se invece raggiunge o supera 1000 il flag è impostato a presente.
2xx	Sottrazione	Sottrae il contenuto della cella di memoria xx dal valore contenuto nell'accumulatore e scrive il valore risultante nell'accumulatore. Il valore salvato nell'accumulatore è la differenza modulo 1000. Se la differenza è inferiore a zero il flag è impostato a presente, se invece è positiva o zero il flag è impostato ad assente.
3xx	Store	Salva il valore contenuto nell'accumulatore nella cella di memoria avente indirizzo xx. Il contenuto dell'accumulatore rimane invariato.
5xx	Load	Scrive il valore contenuto nella cella di memoria di indirizzo xx nell'accumulatore. Il contenuto della cella di memoria rimane invariato.
6xx	Branch	Salto non condizionale. Imposta il valore del program counter a xx.
7xx	Branch if zero	Salto condizionale. Imposta il valore del program counter a xx solamente se il contenuto dell'accumulatore è zero e se il flag è assente.
8xx	Branch if positive	Salto condizionale. Imposta il valore del program counter a xx solamente se il flag è assente.
901	Input	Scrive il contenuto presente nella testa della coda in input nell'accumulatore e lo rimuove dalla coda di input.
902	Output	Scrive il contenuto dell'accumulatore alla fine della coda di output. Il contenuto dell'accumulatore rimane invariato.
0xx	Halt	Termina l'esecuzione del programma. Nessuna ulteriore istruzione viene eseguita.

Alcuni devi valori non corrispondono a nessuna istruzione. Ad esempio tutti i numeri tra 400 e 499 non hanno un corrispettivo. Questo significa che corrispondono a delle istruzioni non valide (illegal instructions) e che LMC si fermerà con una condizione di errore.

Quindi, ad esempio, se l'accumulatore contiene il valore 42, il program counter ha valore 10 ed il contenuto della cella numero 10 è 307, il LMC eseguirà una istruzione di **store**, dato che il valore è nella forma 3xx. In particolare, il contenuto dell'accumulatore, ovvero 42, verrà scritto nella cella di memoria numero 7 (dato che xx corrisponde a 07). Il program counter verrà poi incrementato di uno, assumendo il valore 11. La procedura verrà ripetuta finché non verrà incontrata un'istruzione di halt o un'istruzione non valida.

ASSEMBLY PER LMC

Oltre a fornire una simulazione per il LMC si deve essere anche in grado di passare da un programma scritto in codice assembly per LMC (che verrà successivamente dettagliato) al contenuto iniziale della memoria del LMC, convertendo quindi il codice assembly in codice macchina.

Nel file sorgente assembly ogni riga contiene al più una etichetta ed una istruzione. Ogni istruzione corrisponde al contenuto di una cella di memoria. La prima istruzione assembly presente nel file corrisponderà al valore contenuto nella cella 0, la seconda al valore contenuto nella cella 1, e così via.

Istruzione	Valori possibili per xx	Significato
ADD xx	Indirizzo o etichetta	Esegui l'istruzione di addizione tra l'accumulatore e il valore contenuto nella cella indicata da xx
SUB xx	Indirizzo o etichetta	Esegui l'istruzione di sottrazione tra l'accumulatore e il valore contenuto nella cella indicata da xx
STA xx	Indirizzo o etichetta	Esegue una istruzione di store del valore dell'accumulatore nella cella indicata da xx
LDA xx	Indirizzo o etichetta	Esegue una istruzione di load dal valore contenuto nella cella indicata da xx nell'accumulatore
BRA xx	Indirizzo o etichetta	Esegue una istruzione di branch non condizionale al valore indicato da xx
BRZ xx	Indirizzo o etichetta	Esegue una istruzione di branch condizionale (se l'accumulatore è zero e non vi è il flag acceso) al valore indicato da xx.
BRP xx	Indirizzo o etichetta	Esegue una istruzione di branch condizionale (se non vi è il flag acceso) al valore indicato da xx.
INP	Nessuno	Esegue una istruzione di input
OUT	Nessuno	Esegue una istruzione di output
HLT	Nessuno	Esegue una istruzione di halt
DAT xx	Numero	Memorizza nella cella di memoria corrispondente a questa istruzione assembly il valore xx
DAT	Nessuno	Memorizza nella cella di memoria corrispondente a questa istruzione assembly il valore 0 (equivalente all'istruzione DAT 0).

Ogni riga del file assembly con una istruzione può contenere prima dell'istruzione una etichetta, ovvero una stringa di caratteri usata per identificare la cella di memoria dove verrà salvata l'istruzione corrente. Le etichette possono poi essere utilizzate al posto degli indirizzi (ovvero numeri tra 0 e 99) all'interno del sorgente. Si veda, ad esempio:

Assembly	Cella di memoria	Codice macchina
`BRA LABEL`	1	600

In questo caso l'etichetta presente nella prima riga assume il valore 0 tutte le volte che appare come argomento di un'altra istruzione. Per un esempio già completo si osservi:

Assembly	Cella di memoria	Codice macchina
`LOAD LDA 0`	0	500
`OUT`	1	902
`SUB ONE`	2	208
`BRZ ONE`	3	708
`LDA LOAD`	4	500
`ADD ONE`	5	108
`STA LOAD`	6	300
`BRA LOAD`	7	600
`ONE DAT 1`	8	1

In questo caso ci sono due etichette (`LOAD` e `ONE`) che assumono valore 0 e 8 rispettivamente. Questo corrisponde alla cella di memoria dove la versione convertita in codice macchina dell'istruzione assembly presente nella stessa riga è stata salvata. Si noti inoltre come le etichette possano essere utilizzate prima di venire definite. Se una riga contiene una doppia barra (`//`) tutto il resto della riga deve venire ignorato e considerato come commento. L'assembly è inoltre case-insensitive e possono esserci uno o più spazi tra etichetta e istruzione e etichetta e argomento. Le seguenti istruzioni sono quindi tutte equivalenti:

```
ADD 3
Add 3
add 3 // Questo è un commento
ADD 3 // Aggiunte il valore della cella 3 all'accumulatore
aDD    3
```

Fatte queste premesse, dovrete quindi preparare un predicato ed una funzione che leggano un file e, se non ci sono problemi, producano lo stato iniziale del LMC.

IMPLEMENTAZIONI

IMPLEMENTAZIONE IN PROLOG

L'implementazione del simulatore di LMC in Prolog deve rispettare le restrizioni elencate in questa sezione.

Lo stato del LMC deve essere rappresentato da un termine composto della seguente forma:

```
state(Acc, Pc, Mem, In, Out, Flag).
```

nel caso non sia stata ancora eseguita una istruzione di halt. Nel caso sia stata eseguita invece lo stato deve essere rappresentato usando il funtore `halted_state` invece di `state`:

```
halted_state(Acc, Pc, Mem, In, Out, Flag).
```

Nel termine composto gli argomenti sono i seguenti:

A. `Acc`. Un numero tra 0 e 999 che rappresenta il valore contenuto nell'accumulatore.
B. `Pc`. Un numero tra 0 e 999 che rappresenta il valore contenuto nel program counter.
C. `Mem`. Una lista di 100 numeri tutti compresi tra 0 e 999 che rappresenta il contenuto della memoria del LMC.
D. `In`. Una lista di numeri tra 0 e 999 che rappresenta la coda di input del LMC.
E. `Out`. Una lista di numeri tra 0 e 999 che rappresenta la coda di output del LMC.
F. `Flag`. Può assumere solo i valori flag e noflag, che indicano rispettivamente che il flag è presente o assente.

Per eseguire il codice del LMC deve essere presente il seguente predicato:

```
one_instruction(State, NewState)
```

ove `State` e `NewState` sono stati del LMC rappresentati come descritto sopra ed il predicato è vero quando l'esecuzione di *UNA singola istruzione* a partire da `State` porta allo stato `NewState`. Il predicato fallisce nei seguenti casi:

A. Lo stato `State` è un `halting_state`, ovvero il sistema è stato arrestato e non può eseguire istruzioni.
B. L'istruzione da eseguire è di input ma la coda di input è vuota.
C. L'istruzione da eseguire non è valida.

In tutti gli altri casi una query della forma `one_instruction(State, X)` dove `State` è fissato e `X` è una variabile deve avere successo e il valore di `X` deve essere l'unico stato che segue allo stato fornito dopo aver eseguito l'istruzione puntata dal program counter.

Si osservi il seguente esempio:

```
?- one_instruction(state(32, 0, [101, 10, 0…0], [], [], noflag), X).
X = state(42, 1, [101, 10, 0…0], [], [], noflag)
```

Dove 0…0 rappresenta i 98 zero che servono ad ottenere una lista di 100 elementi. Nello stato iniziale fornito l'istruzione da eseguire è quella in posizione 0 (dato che il program counter ha valore 0). Questa istruzione ha valore numerico 101, ovvero è una istruzione di addizione che somma al valore contenuto nell'accumulatore il valore contenuto nella cella 1. In questo caso i due valori da sommare sono 32 (accumulatore) e 10 (cella di memoria 1). Il risultato, 42, diventa il nuovo valore dell'accumulatore ed il valore del program counter viene incrementato di uno. Notare che, dato che il valore della somma non supera 999 il flag rimane spento (`noflag`).

Deve essere inoltre presente il seguente predicato:

```
execution_loop(State, Out)
```

ove `State` rappresenta lo stato iniziale del LMC e `Out` la coda di output nel momento in cui viene raggiunto uno stato di stop (e quindi eseguita una istruzione di halt). Il predicato deve fallire nel caso l'esecuzione termini senza eseguire una istruzione di halt (ad esempio se si incontra una istruzione non valida).

Infine, dovrete produrre due predicati per la gestione dell'assembly. Il primo è un predicato dal nome `lmc_load/2` che si preoccupa di leggere un file che contiene un codice assembler e che produce il contenuto "iniziale" della memoria sistema (una lista di 100 numeri tra 0 e 999).

```
lmc_load(Filename, Mem)
```

dove `Filename` è il nome di un file e `Mem` è la memoria del sistema nel suo "stato iniziale".

Il secondo è un predicato dal nome `lmc_run/3` che si preoccupa di leggere un file che contiene un codice assembler, lo carica (con `lmc_load/2`), imposta la coda di input al valore fornito e produce un output che è il risultato dell'invocazione di `execution_loop/2`.

```
?- lmc_run("my/prolog/code/lmc/test-assembly-42.lmc", [42], Output)
```

è un esempio della sua invocazione.

Suggerimenti

Per implementare al meglio (ed in modo semplice) il simulatore LMC in Prolog vi consigliamo come minimo, di utilizzare il predicato `nth0/4`.

Un modo per generare uno "stato" iniziale è di usare il predicato di libreria `randseq/3`; attenzione che il primo argomento non può essere superiore al secondo. Ad esempio:

```
?- randseq(99, 99, Mem).
Mem = [33, 10, 64, 29, 62, 53, 98, 22, 36|...].
```

IMPLEMENTAZIONE IN COMMON LISP

L'implementazione del simulatore di LMC in Common Lisp deve rispettare le restrizioni elencate in questa sezione.

Lo stato del LMC deve essere rappresentato da una lista della seguente forma:

```
(STATE :ACC <Acc> :PC <PC> :MEM <Mem> :IN <In> :OUT <Out> :FLAG <Flag>)
```

nel caso non sia stata ancora eseguita una istruzione di halt. Nel caso sia stata eseguita invece lo stato deve essere rappresentato usando il funtore `HALTED-STATE` invece di `STATE`:

```
(HALTED-STATE :ACC <Acc>
              :PC <PC>
              :MEM <Mem>
              :IN <In>
              :OUT <Out>
              :FLAG <Flag>)
```

Nella lista qui sopra gli argomenti sono i seguenti:
A. `Acc`. Un numero tra 0 e 999 che rappresenta il valore contenuto nell'accumulatore.

B. `Pc`. Un numero tra 0 e 999 che rappresenta il valore contenuto nel program counter.
C. `Mem`. Una lista di 100 numeri tutti compresi tra 0 e 999 che rappresenta il contenuto della memoria del LMC.
D. `In`. Una lista di numeri tra 0 e 999 che rappresenta la coda di input del LMC.
E. `Out`. Una lista di numeri tra 0 e 999 che rappresenta la coda di output del LMC.
F. `Flag`. Può assumere solo i valori `flag` e `noflag`, che indicano rispettivamente che il flag è presente o assente.

Per eseguire il codice del LMC deve essere definita la seguente funzione:

```
one-instruction State ==> NewState
```

ove `State` e `NewState` sono stati del LMC rappresentati come descritto sopra ed il predicato è vero quando l'esecuzione di *UNA singola istruzione* a partire da `State` porta allo stato `NewState`. Il predicato fallisce nei seguenti casi:

A. Lo stato `State` è un `HALTING-STATE`, ovvero il sistema è stato arrestato e non può eseguire istruzioni.
B. L'istruzione da eseguire è di input ma la coda di input è vuota.
C. L'istruzione da eseguire non è valida.

In tutti gli altri casi l'invocazione della funzione `(one-instruction State)` dove `State` è fissato ritorna un nuovo stato `NewState` che deve essere l'unico stato che segue allo stato fornito dopo aver eseguito l'istruzione puntata dal program counter.

Si osservi il seguente esempio:

```
?- (one-instruction '(state :acc 32
                            :pc 0
                            :mem (101 10 0…0)
                            :in ()
                            :out ()
                            :flag noflag))
(STATE :ACC 42 :PC 1 :MEM (101 10 0…0) :IN () :OUT () :FLAG NOFLAG)
```

Dove 0…0 rappresenta i 98 zero che servono ad ottenere una lista di 100 elementi. Nello stato iniziale fornito l'istruzione da eseguire è quella in posizione 0 (dato che il program counter ha valore 0). Questa istruzione ha valore numerico 101, ovvero è una istruzione di addizione che somma al valore contenuto nell'accumulatore il valore contenuto nella cella 1. In questo caso i due valori da sommare sono 32 (accumulatore) e 10 (cella di memoria 1). Il risultato, 42, diventa il nuovo valore dell'accumulatore ed il valore del program counter viene incrementato di uno. Notare che, dato che il valore della somma non supera 999 il flag rimane spento (`noflag`).

Deve essere inoltre presente il seguente predicato:

```
execution-loop State ==> Out
```

ove `State` rappresenta lo stato iniziale del LMC e `Out` la coda di output nel momento in cui viene raggiunto uno stato di stop (e quindi eseguita una istruzione di halt). La funzione deve generare un errore nel caso l'esecuzione termini senza eseguire una istruzione di halt (ad esempio se si incontra una istruzione non valida).

Infine, dovrete produrre due funzioni per la gestione dell'assembly. La prima è una funzione dal nome `lmc-load` che si preoccupa di leggere un file che contiene un codice assembler e che produce il contenuto "iniziale" della memoria del sistema (una lista di 100 numeri tra 0 e 999).

```
lmc-load Filename ==> Mem
```

dove `Filename` è il nome di un file e `Mem` è il contenuto della memoria nello "stato iniziale" del sistema.

La seconda è una funzione dal nome `lmc-run` che si preoccupa di leggere un file che contiene un codice assembler, lo carica (con `lmc-load`), inizializza la coda di input al valore fornito, e produce un output che è il risultato dell'invocazione di `execution-loop`.

```
cl-prompt> (lmc-run "my/common-lisp/code/lmc/test-assembly-42.lmc", '(9))
;;; Output
```

è un esempio della sua invocazione.

Suggerimenti

Per implementare al meglio (ed in modo semplice) il simulatore LMC in Lisp vi permettiamo di usare la funzione `setf` in combinazione con la funzione `nth`.

Un modo per generare uno "stato" iniziale è di usare la funzione `make-list`, as in

```
cl-prompt> (make-list 100 :initial-element 0)
(0 0 0 0 …) ; 100 zeri.
```

ISTRUZIONI PER LA CONSEGNA

Dovrete consegnare un file .zip che contiene l'intero progetto. Leggete attentamente le istruzioni per la consegna.

Ripetiamo: LEGGETE MOLTO, MA MOLTO ATTENTAMENTE LE ISTRUZIONI PER LA CONSEGNA!!!!

LE CONSEGNE CHE NON RISPETTERANNO LE ISTRUZIONI SARANNO SEMPLICEMENTE IGNORATE CON CONSEGUENTE VOTO NEGATIVO.

A questo punto ripetiamo e specifichiamo!

La data di consegna è DOMENICA 13 GENNAIO 2018, ORE 23:55 Zulu Time (Solare)

Dovrete consegnare un file `.zip` (i files `.7z`, `.rar` o `.tar` etc, *non sono accettabili*!!!) dal nome

```
MATRICOLA_Cognome_Nome_LP_E1P_2018_LMC.zip
```

Nome e *Cognome* devono avere solo la prima lettera maiuscola, *Matricola* deve avere lo zero iniziale se presente. Cognomi e nomi multipli vanno inframmezzati con il carattere '_'; ad esempio: `Pravettoni_Brambilla_Gian_Giac_Pier_Carluca`.

Questo file compresso deve contenere una sola directory con lo stesso nome. Al suo interno ci deve essere una sottodirectory chiamata '`Prolog`' e una sottodirectory chiamata '`Lisp`'. Al loro interno queste directory devono contenere i files caricabili e interpretabili, più tutte le istruzioni che riterrete necessarie. Il file Prolog si deve chiamare '`lmc.pl`', e il file Lisp si deve chiamare '`lmc.lisp`'. Le due sottodirectory devono contenere un file chiamato `README.txt`. In altre parole questa è la struttura della directory (folder, cartella) una volta spacchettata.

```
424241_Pravettoni_Brambilla_Gian_Giac_Pier_Carluca_LP_E1P_2018_LMC
      Prolog
            lmc.pl
            README.txt
      Lisp
            lmc.lisp
            README.txt
```

Potete aggiungere altri files, ma il loro caricamento dovrà essere effettuato automaticamente al momento del caricamento ("loading") dei files sopracitati.
Come sempre, valgono le direttive standard (reperibili sulla piattaforma Moodle) circa la formazione dei gruppi; ovvero, massimo 3 persone per gruppo.
Ogni file deve contenere all'inizio un commento con il nome e matricola di ogni membro del gruppo. Ogni persona deve consegnare un elaborato, anche quando ha lavorato in gruppo.

VALUTAZIONE

In aggiunta a quanto detto precedentemente seguono ulteriori informazioni sulla procedura di valutazione.

Abbiamo a disposizione una serie di esempi standard che saranno usati per una valutazione oggettiva dei programmi.

Se i files sorgente non potranno essere letti/caricati nell'ambiente Prolog (nb.: SWI-Prolog, ma non necessariamente in ambiente Windows, Linux, Mac), o nelll'ambiente Common Lisp (Lispworks, ma non necessariamente in ambiente Windows, Linux, Mac), il progetto non sarà ritenuto sufficiente.

Il mancato rispetto dei nomi indicati per funzioni e predicati, o anche delle strutture proposte e della semantica esemplificata nel testo del progetto, oltre a comportare ritardi e possibili fraintendimenti nella correzione, può comportare una diminuzione nel voto ottenuto o alla sua bocciatura.

Errata e chiarimenti

20 novembre 2018

Il program counter contiene numeri tra 0 e 99 (e non 999, come indicato nel testo). Se si incrementa un program counter contenente il valore 99 il valore successivo sarà 0.

24 novembre 2018

Se il programma assembly contiene più di 100 istruzioni il predicato lmc_load deve fallire. Un programma assembly può contenere meno di 100 istruzioni: la parte di memoria che il programma non occupa contiene il valore 0. Inoltre, i file assembly forniti in input possono anche essere non sintatticamente corretti (in quel caso il predicato lmc_load deve fallire)-

30 novembre 2018

Si può assumere che le etichette non abbiano lo stesso nome delle istruzioni. Nel file assembly, inoltre, potrebbe non esserci uno spazio tra istruzione e commento.
Nel testo del progetto tutte le volete che compare “halting-state” si legga invece “halted-state”.

2 dicembre 2018

I nomi validi delle etichette sono una qualsiasi sequenza alfanumerica e per semplicità si può assumere che inizi per una lettera. Se il programma riconosce altri nomi in aggiunta a questi non ci sono problemi.
L’istruzione DAT può avere come argomento un qualsiasi numero tra 0 e 999.