Gestione di Liste Circolari in RISC-V

Progetto di Architettura degli Elaboratori 22-23

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Matteo Becatti
matteo.becatti@stud.unifi.it
7006081

August 15, 2023

Contents

1

Parsing

1.1 Parsing dell'input

Gli input sono passati al programma tramite la stringa *listInput*, che contiene le varie istruzioni, separate l'una da l'altra con una tilda.

Il primo passo è controllare se sono presenti spazi all'inizio della stringa, utilizzando *check_ initial_ spaces*. Finchè sono presenti spazi, li ignoriamo e passiamo al carattere successivo.

I caratteri vengono letti uno alla volta. Una versione semplificata del processo eseguito dal programma è illustrata nell'immagine seguente:

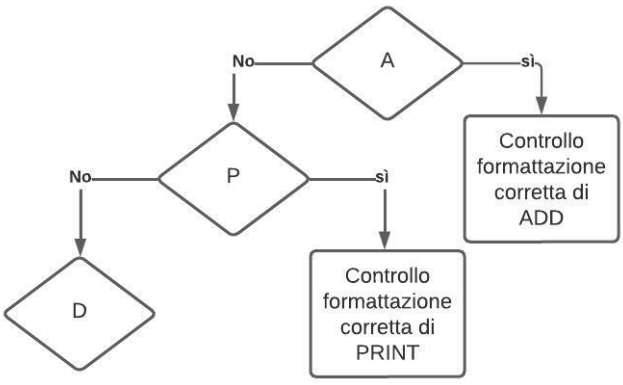

Figure 1.1: Controllo dei caratteri

Leggiamo il carattere e chiediamo "È la lettera A?" Se sì, procediamo con i controlli di formattazione per A, se la risposta è no, ci chiediamo "È la lettera

P?" e così via per tutte le altre iniziali. Per ogni operazione viene seguito un processo molto simile, e le differenze tra di loro possono essere divise in tre categorie. I nomi dei controlli seguono la forma *check_ nomeistruzione*, es: *check_ add* per ADD.

Tutti i controlli finiscono con un jump-and-link verso la rispettiva istruzione della lista. Quando l'istruzione sarà completa, tramite il return address il program counter tornerà nel punto da cui era stato chiamato. La riga successiva è sempre *"j check_ next_ instruction"*

1.2 ADD, DEL

Il parsing di ADD e DEL è identico, quindi per la spiegazione prendiamo ADD come esempio. La sintassi corretta deve seguire la forma ADD(n), dove n è l'elemento da aggiungere alla lista, e può essere solo un singolo carattere. Una volta letta la A, proseguiamo verso i caratteri successivi, facendo i necessari controlli. Leggiamo D due volte, e poi controlliamo che ci sia una parentesi aperta. Ora dobbiamo leggere il valore da aggiungere alla lista. Lo salviamo nel registro a2, in modo che sia ancora disponibile quando chiameremo l'istruzione ADD alla fine dei controlli, e verifichiamo che rientri in un range di valori ASCII accettabile (non tutti i valori sono ammessi). Ci assicuriamo che l'istruzione sia formattata correttamente con la parentesi chiusa alla fine, e saltiamo all'istruzione ADD per aggiungere il nuovo elemento alla lista.

1.2.1 jump-and-link

Alla fine del procedimento, e dopo aver controllato che non ci siano spazi vuoti tramite la procedura *check_ spaces*, viene effettuato un *jump-and-link* verso l'istruzione ADD. L'elemento da aggiungere viene passato come parametro, e la posizione del program counter è automaticamente salvata nel return address. Non ci aspettiamo valori di ritorno. Quando la ADD sarà conclusa, torneremo all'indirizzo salvato nel ra, e il passo successivo sarà saltare verso *check_ next_ instruction*. Questo procedimento è uguale anche per DEL, ovviamente in questo caso il parametro non è l'elemento da aggiungere, ma quello da rimuovere. Per le altre funzioni invece non vengono passati parametri.

1.3 PRINT, REV

PRINT e REV sono più semplici di ADD e DEL. Effettuano il controllo sulla formattazione delle lettere, ma non avendo parametri da dover gestire, saltano direttamente alla rispettiva istruzione, senza bisogno di dover toccare i registri per salvare dati.

1.4 SORT, SDX, SSX

Fino ad ora le istruzione avevano tutte iniziali differenti. SORT, SDX e SSX invece iniziano tutte con S. Questa volta il programma effettua prima un controllo su S, e poi verifica quale è la lettera successiva, ed in base a quello che legge procede verso la rispettiva procedura. Anche in questo caso non ci sono parametri da dover gestire, quindi saltiamo direttamente alla procedura corrispondente.

1.5 Formattazione errata

Il programma è in grado di gestire le situazioni in cui la stringa di input non è formattata correttamente. Se ci sono spazi vuoti all'inizio o alla fine della stringa, oppure prima o dopo una tilda, verranno ignorati. Se sono presenti spazi all'interno dell'istruzione invece, essa verrà scartata. Durante la lettura di una istruzione, appena troviamo un errore di scrittura (ad esempio PRNT anzichè PRINT), scartiamo il comando e passiamo al prossimo. Nel caso delle istruzioni con parametri, se viene inserito più di un carattere dentro alle parentesi, la formattazione è considerata errata.

2

Funzioni

2.1 Funzioni del programma

Una volta completato il parsing di una singola istruzione, viene effettuato un salto verso la rispettiva funzione. Ad esempio, quando *check_add* finisce con successo, salta verso *ADD*.

2.2 ADD

La funzione di ADD si occupa di inserire nuovi elementi all'interno della lista. Inizia chiamando *address_generator*, che si occupa di generare l'indirizzo di memoria per il nodo che dobbiamo aggiungere

Una volta ottenuto l'indirizzo, controlliamo se l'operazione che stiamo facendo è la prima del suo tipo, poichè:

- Se la lista è vuota, aggiungiamo il nuovo nodo, lo sincronizziamo con la testa globale, e impostiamo il suo PAHEAD (il puntatore al nodo successivo) verso se stesso.
- Se la lista non è vuota, e quindi c'è almeno un elemento, impostiamo il puntatore del nuovo nodo verso la testa, ma dobbiamo anche preoccuparci di collegare la coda con il nuovo nodo. Per farlo, chiamiamo la procedura *get_last_node*, che ci restituisce l'ultimo nodo della lista, e muoviamo il suo puntatore verso il nodo appena aggiunto.

2.2.1 jal address_generator

Per usare il generatore di indirizzi, abbiamo bisogno di effettuare un jal. Il problema è che abbiamo già fatto un jal dal parsing verso la funzione ADD, e

facendone un altro andremo a perdere il return address. Possiamo utilizzare lo stack per risolvere questo problema. Creiamo uno stack di dimensione 4 byte e ci carichiamo il return address. Chiamiamo la *jal address_ generator*, e al riprendiamo il return address dalla stack, e poi la resettiamo. Una illustrazione del comportamento dello stack pointer può essere vista in figura. Non essendoci bisogno di salvare parametri nello stack, il suo contenuto e le rispettive operazioni risultano molto basiche.

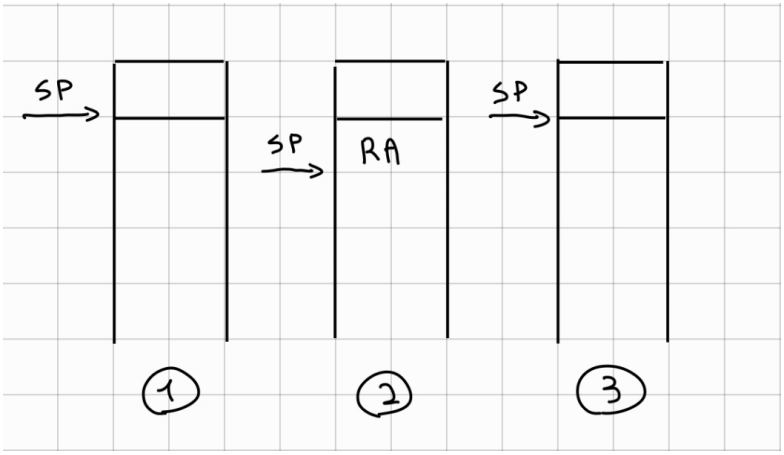

Figure 2.1: Stack prima, durante e dopo la jal

2.2.2 jal get_last_node

Nel corso del codice, abbiamo spesso bisogno di accedere all'ultimo elemento, ma le specifiche non permettono di salvare un puntatore globale verso la coda. Ho dunque scritto una funzione che cicla attraverso la lista e restituisce la coda. Anche qua però si pone lo stesso problema del generatore di indirizzi, dobbiamo effettuare un *jump_ and_ link* in una procedura interna. e la soluzione è sempre la stessa. Questa volta però abbiamo anche un parametro di ritorno, infatti il nodo coda viene restituito tramite il registro a0.

Questa chiamata si trova numerose volte all'interno del codice, e non sarà spiegata nuovamente.

2.2.3 Generatore di indirizzi di memoria

Genera un indirizzo di memoria casuale a partire da un seed fornito. La procedura effettua 4 shift a destra del seed (di 1, 2^2, 2^3 e 2^5) e salva i risultati in 4 registri temporanei diversi. Successivamente, questi risultati vengono combinati tramite una funzione XOR e il seed viene shiftato a destra di 2.

Il risultato dello XOR viene shiftato a sinistra di 2^(15) e questi due valori vengono combinati tramite una funzione OR. Per ottenere solo i primi 16 bit del risultato dell'OR, si applica una funzione AND con il valore 0x0000ffff. L'output di questa operazione viene combinato tramite una funzione OR con il valore 0x00010000 per ottenere l'indirizzo generato. Questo indirizzo viene sovrascritto al seed precedente in caso di una futura chiamata della procedura.

Infine, la procedura controlla che la memoria puntata dall'indirizzo sia libera. In caso positivo, la funzione termina e restituisce l'indirizzo generato. In caso contrario, la procedura si richiama ricorsivamente per generare un nuovo indirizzo casuale."

2.3 PRINT

PRINT stampa il contenuto della lista, mostrando gli elementi nello stesso ordine in cui si trovano all'interno della struttura. Inoltre aggiunge delle parentesi quadre per aumentare la chiarezza della lettura. Anche se la lista è vuota, verranno stampate delle parentesi per mostrare che la stampa è stata effettuata, ma non ci sono elementi nella lista.

Il puntatore alla testa della lista viene copiato in un registro temporaneo, dandoci la possibilità di iterare senza dover modificare il puntatore originale. Tramite un controllo verifichiamo che sia stata effettuata almeno una operazione di ADD, in caso negativo non ci sarebbe niente da stampare, ed entrare nel ciclo sarebbe superfluo, dunque terminiamo l'esecuzione di PRINT per passare alla funzione successiva.

La stampa avviene all'interno di un ciclo a contatore, che continua finchè non raggiungiamo il numero di elementi della lista. Ogni volta carichiamo in a0 il carattere da stampare, ed effettuiamo la chiamata di sistema. Alla fine della lista, chiudiamo la parentesi quadra e andiamo a capo.

2.4 DEL

Elimina un elemento dalla lista. Controlla se è stata fatta almeno una ADD, ed in caso negativo termina l'operazione e passa alla successiva. L'eliminazione avviene all'interno di un ciclo che scorre la lista finchè non trova un elemento che corrisponda al parametro passato in input, oppure arriva alla fine della lista e termina nel caso in cui l'elemento non sia presente.

Una volta trovato l'elemento, teniamo conto della sua posizione. Se il nodo si trova...

- **in testa** alla lista, si fa una jump a *del_first_element*. La memoria occupata dal nodo viene liberata, permettendo al generatore di indirizzi di poterla riusare nell'eventualità in cui venga selezionata di nuovo. Il puntatore alla testa della lista viene spostato verso quello che era il secondo elemento.

- **in una posizione intermedia** saltiamo a *del_elemento*. Il PAHEAD del nodo precedente va impostato al nodo successivo rispetto a quello che vogliamo eliminare, e azzeriamo il contenuto del nodo.

- è **l'unico elemento**, liberiamo la memoria utilizzata dal nodo ed eliminiamo il puntatore. Il puntatore alla testa globale viene a sua volta azzerato.

2.5 REV

Inverte la lista esistente.

Utilizzando due indici, uno che scorre la lista partendo dalla testa (sinistra) e uno dalla coda (testa), scambiamo gli elementi, finchè non troviamo una collisione tra gli indici. Si possono presentare due casi:

- Elementi pari: Gli indici si incontreranno e si sorpasseranno, terminando l'esecuzione.

- Elementi dispari: Gli indici si incontreranno nell'elemento nel mezzo alla lista, la cui posizione rimarrà invariata. L'esecuzione termina.

Se la lista fosse bidirezionale e con possibilità di salvare il puntatore della coda, l'esercizio sarebbe finito qua. Invece per funzionare c'è bisogno di fare qualche operazione in più. All'inizio di ogni iterazione di REV, c'è un ulteriore ciclo che si occupa di scorrere la lista finchè non raggiunge la posizione del contatore del nodo destro Questo contatore diminuisce ad ogni iterazione poichè deve scorrere verso sinistra, e ci permette di prelevare il nodo più a sinistra non ancora scambiato.

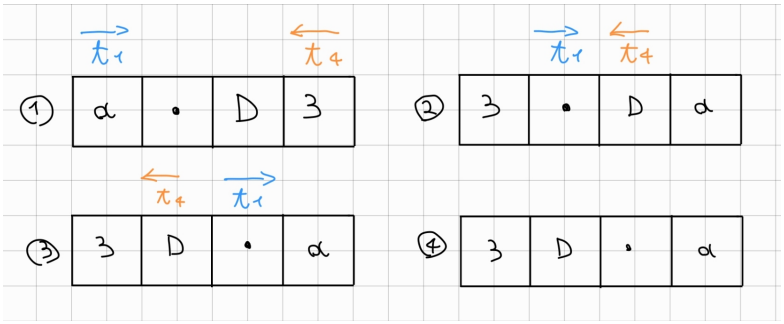

Figure 2.2: Esempio di REV per una lista con un numero di elementi pari. È visualizzata come un vettore per motivi di chiarezza.

2.6 SORT

L'ordinamento della lista è senza dubbio la funzione più complessa presente nel codice.

Ho deciso di usare il Bubble Sort, un algoritmo di ordinamento che confronta e scambia gli elementi adiacenti di una lista se sono nell'ordine sbagliato. La variante con flag usa una variabile booleana per verificare se la lista è già ordinata e interrompere l'algoritmo se non ci sono stati scambi nell'ultima iterazione.

Inizialmente la flag è impostata a 0, poichè ancora non ci sono stati spostamenti. Carichiamo gli elementi del nodo attuale e del nodo successivo per il confronto. Se il successivo è la testa, sappiamo che siamo arrivati all'ultimo elemento, e fermiamo il SORT_loop, altrimenti procediamo con i vari controlli all'interno di swap_check. Se swap_check mette la flag a true, saltiamo su swap_element, che come indito dal nome, scambia gli elementi. Una volta che il ciclo ha esaminato tutti gli elementi della lista, SORT viene chiamato di nuovo, e il ciclo riparte dall'inizio. SORT verrà chiamato finchè tutti gli elementi non sono ordinati.

2.6.1 swap_check

Si occupa di confrontare i due elementi presi in esame dal Bubble Sort. Il progetto richiede un certo tipo di priorità: *carattere extra* < *numero* < *letter minuscola* < *lettera maiuscola*. Ognuna di queste categoria appartiene ad un range di valori ASCII (es: $65 \leq maiuscola \leq 90$). La procedura non svolge niente di particolarmente elaborato, fa solo una serie di "less then" e "greater then" per capire all'interno di quale gruppo appartengono i due

numeri da confrontare, e se risulta che debbano essere scambiati, imposta la flag a2 ad 1.

2.7 SDX

L'implementazione di SDX è molto semplice. Tramite la procedura vista in precedenza otteniamo la coda. Assegnamo al puntatore globale la coda. Ora la lista è stata spostata di una posizione verso destra.

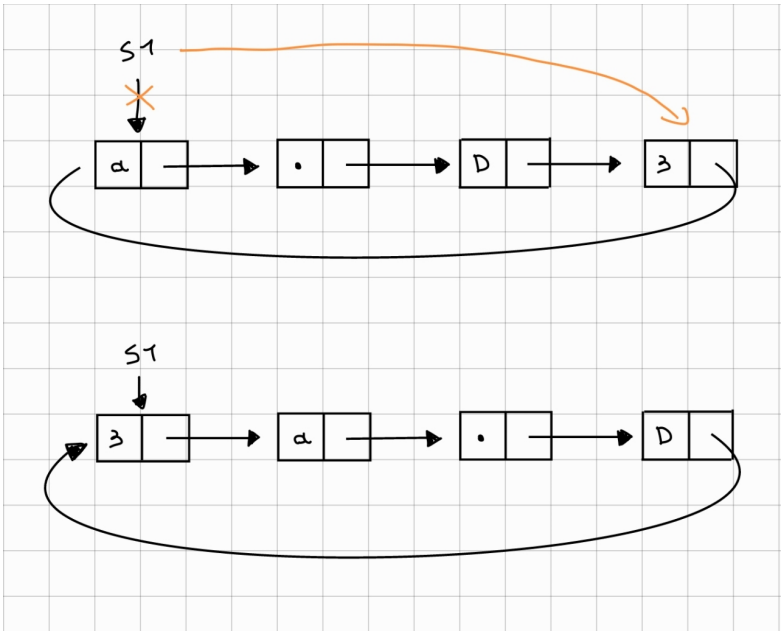

Figure 2.3: Shift a destra degli elementi della lista

2.8 SSX

Lo shift a sinistra è ancora più semplice di quello a destra, poichè non dobbiamo andare a recuperare l'ultimo elemento della lista. Carichiamo la testa, leggiamo l'elemento successivo e lo assegniamo al puntatore globale della testa (s1). Ora gli elementi sono stati spostati verso sinistra di una posizione.

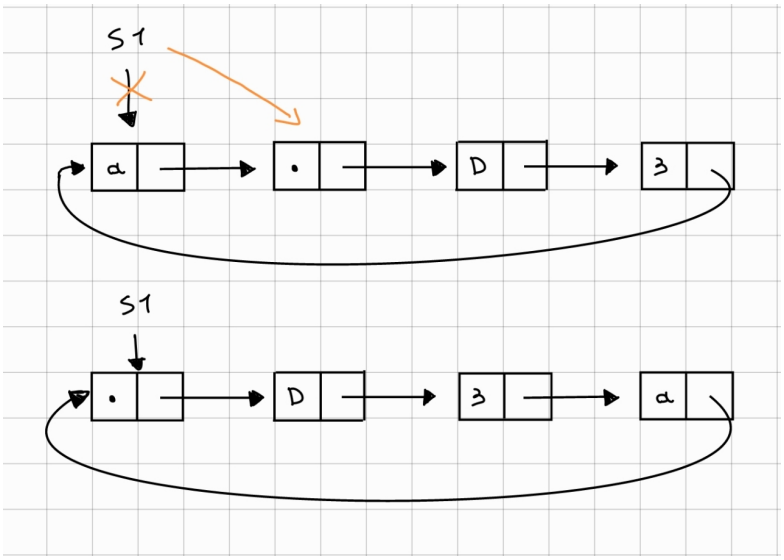

Figure 2.4: Shift a sinistra degli elementi della lista

3

Testing

Durante lo sviluppo, ho utilizzato diversi tipi di stringhe per verificare il corretto funzionamento del codice. Queste stringhe erano inizialmente molto semplici, e man mano che tutto sembrava funzionare correttamente, le rendevo più complesse per verificare i casi limite.

Sono arrivato alla conclusione che per verificare il corretto funzionamento del programma, ognuna delle operazioni della lista dovesse funzionare nei seguenti casi:

- Non ci sono elementi nella lista
- C'è un solo elemento della lista
- Ci sono due elementi nella lista
- C'è un numero pari di elementi nella lista
- C'è un numero dispari di elementi nella lista
- Ci sono errori di formattazione nel codice

Per questo motivo, nella sezione .data del programma, oltre alle due stringhe di test fornite nelle specifiche del progetto, ho aggiunto una stringa per ogni funziona che testasse queste situazioni. Non è stata aggiunta una stringa per PRINT, poichè tutti i suoi casi di funzionamento sono visibili all'interno dei test delle altre funzioni.

Gli input sono visibili nell'immagine sottostante. Ogni riga è stata spezzata in due per favorire la leggibilità, nel codice originale ogni input si trova su una sola riga.

```
.data
listInput: .string "ADD(1) ~ ADD(a) ~ ADD(a) ~ ADD(B) ~ ADD(;) ~
    ADD(9) ~SSX~SORT~PRINT~DEL(b)~DEL(B) ~PRI~SDX~REV~PRINT"

# listInput: .string "ADD(1) ~ SSX ~ ADD(a) ~ add(B) ~ ADD(B) ~ ADD
# ~ ADD(9) ~PRINT~SORT(a)~PRINT~DEL(bb)~DEL(B) ~PRINT~REV~SDX~PRINT"

# Test ADD
# listInput: .string "ADD(a)~PRINT~ADD(b)~PRINT~ADD(c)~PRINT~ADD(d)~
# ~ADD(e)~~ADD(f)~~ADD(g)~~ADD(h)~~ADD(i)~~ADD(j)~PRINT~ADD(k)~PRINT"

# Test DEL
# listInput: .string "ADD(a)~PRINT~DEL(a)~PRINT~ADD(a)~ADD(b)~ADD(a)~PRINT
# ~DEL(a)~DEL(b)~DEL(c)~PRINT~ADD(a)~ADD(b)~ADD(b)~ADD(c)~PRINT~DEL(b)~PRINT "

# Test REV
# listInput: .string "REV~PRINT~ADD(a)~PRINT~REV~ADD(c)~ADD(x)~ADD(b)
# ~ADD(h)~PRINT~REV~PRINT~DEL(c)~PRINT~REV~PRINT"

# Test SSX
# listInput: .string "SSX~PRINT~ADD(a)~PRINT~SSX~ADD(c)~ADD(x)~ADD(b)
# ~ADD(h)~PRINT~SSX~PRINT~DEL(c)~PRINT~SSX~PRINT"

# Test SDX
# listInput: .string "SDX~PRINT~ADD(a)~PRINT~SDX~ADD(c)~ADD(x)~ADD(b)
# ~ADD(h)~PRINT~SDX~PRINT~DEL(c)~PRINT~SDX~PRINT"

# Test SORT
# listInput: .string "SORT~PRINT~ADD(a)~PRINT~SORT~ADD(C)~ADD(.)~ADD(b)
# ~ADD(1)~PRINT~SORT~PRINT~ADD(b)~ADD(b)~ADD(b)~PRINT~SORT~PRINT"

# Test formattazione errata
# listInput: .string "ADD(a)~PRIT~add(b)~PRINT~AD D(c)~PRINT~ADD(d)~rev
# ~ADD(f)~SOLT~ADD(h)~SdX~ADD(j)~PRINT~ADD(k)~PRINT"

```

Figure 3.1: I listInput presenti nel codice

3.1 Risultati dei test

I risultati dei test visti nell'immagine sopra sono visibili di seguito:

```
# Primo input delle specifiche
[ ; 1 9 a a B ]
[ a 9 1 ; a ]
# Secondo input delle specifiche
[ 1 a B 9 ]
[ 1 9 a B ]
[ 1 9 a ]
[ 1 a 9 ]
# Risultato test ADD
[ a ]
[ a b ]
[ a b c ]
[ a b c d e f g h i j ]
[ a b c d e f g h i j k ]
# Risultato test DEL
[ a ]
[ ]
[ a b a ]
[ ]
[ a b b c ]
[ a c ]
# Risultato test REV
[ ]
[ a ]
[ a c x b h ]
[ h b x c a ]
[ h b x a ]
[ a x b h ]
# Risultato test SSX
[ ]
[ a ]
[ a c x b h ]
[ c x b h a ]
[ x b h a ]
[ b h a x ]
# Risultato test SDX
[ ]
[ a ]
[ a c x b h ]
[ h a c x b ]
```

```
[ h a x b ]
[ b h a x ]
# Risultato test SORT
[ ]
[ a ]
[ a C . b 1 ]
[ . 1 a b C ]
[ . 1 a b C b b b ]
[ . 1 a b b b b C ]
# Risultato test formattazione errata
[ a ]
[ a ]
[ a d f h j ]
[ a d f h j k ]
```

4

Note finali

4.0.1 Lettere accentate problematiche

Il codice è stato scritto utilizzando Linux. Questo non comporta problemi per le funzionalità del progetto. Il problema sorge nei commenti, poichè Linux di default utilizza la codifica UTF-8 per il testo, mentre Windows no. Ho provato il codice su un PC con Windows 11, e tutte le lettere accentate erano state rimpiazzate con un carattere di errore. Ho risolto questo problema impostando anche su Windows la codifica UTF-8.

4.0.2 Ripes

Sia su Linux che su Windows è sempre stata utilizzata quella che ad oggi è l'ultima release stabile di Ripes, la Release build (v.2.2.6) del 7 Gennaio 2023, scaricata dalla repository Github del progetto.