Michele Sbicego **FBK**

Manuale

Indice

1 Notazione

$
- indica che ci troviamo sul terminale

>
- indica il passaggio ad una sottocartella

→
- indica il passaggio ad un'azione successiva

+
- indica la pressione contemporanea di piu' tasti

⟨ ⟩
- il contenuto delle parentesi angolate e' a titolo di esempio e puo' variare

[ ]
- il contenuto delle parentesi quadre e' facoltativo, a parte nelle occorrenze in cui ne viene specificata esplicitamente una semantica

2 Generale

→ BANDIERINA+⟨arrow⟩
- modifica la posizione e le dimensioni della finestra corrente a seconda della freccia cliccata:

freccia destra : posiziona a destra,
freccia sinistra : posiziona a sinistra,
freccia su : a tutto schermo,
freccia giu' : ridimensiona ai valori di default.

→ BANDIERINA+H
- minimizza la finestra attuale

2.1 Percorsi

Sono supportati i percorsi di altre macchine. La sintassi e':

/ssh:⟨user⟩@⟨machine⟩:⟨path⟩

3 Installazione

Quando si vuole installare un'applicazione, solitamente si apre il terminale nella cartella sorgente dell'applicazione e si danno i seguenti comandi:

- ./configure
- make

- `make install`

L'installazione avviene nella ROOT della propria macchina, solitamente nel path `/usr`. Se non si hanno diritti di amministratore sulla propria macchina bisogna installare l'applicazione nella propia HOME. Solitamente i comandi da dare sono:

- `./configure --prefix=$HOME/usr`
- `make`
- `make install`

Successivamente, bisogna aggiungere l'informazione del nuovo path a bash. Bisogna aggiungere la riga:

```
export PATH=``$HOME/usr/bin:$PATH''
```

nel file `$HOME/.bash`. Facoltativamente, si puo' dare un soprannome all'eseguibile per dare la priorita' all'eseguibile nella HOME rispetto a quello nella ROOT. Bisogna aggiungere la riga:

```
alias ⟨exe⟩=``$HOME/usr/bin/⟨exe⟩''
```

nel file `$HOME/.bash`.

4 Terminale

Descrizione: il terminale consente di gestire i file e i processi.

4.1 Istanze

$ `Ctrl + Shift + T`
- apre un nuovo terminale nella finestra terminale corrente

$ `Ctrl + Shift + PageUp`
- seleziona il terminale successivo della finestra terminale corrente

$ `Ctrl + Shift + PageDown`
- seleziona il terminale precedente della finestra terminale corrente

4.2 Cartelle e File

$ `pwd`
- stampa la posizione assoluta attuale

$ `ls`
- stampa gli oggetti presenti nella cartella attuale

$ `ll`
- stampa gli oggetti presenti nella cartella attuale, con il tipo di accesso che permettono

$ `ls -a`
- stampa gli oggetti presenti nella cartella attuale, compresi quelli nascosti

$ `cd ⟨path⟩`

- sposta la posizione attuale nella cartella specificata

$ `rm ⟨file⟩`

- cancella il file

$ `rm -rf ⟨folder⟩`

- cancella la cartella

$ `cp [⟨path⟩/]⟨file⟩ [⟨path⟩/]⟨new_file⟩`

- copia e incolla

$ `mv [⟨path⟩/]⟨file⟩ [⟨path⟩]⟨new_file⟩`

- taglia e incolla con un nuovo nome – come caso particolare si ottiene la ridenominazione

$ `scp [⟨machine_name⟩:][⟨path⟩/]⟨file⟩ [⟨machine_name⟩:][⟨path⟩]⟨new_file⟩`

- copia e incolla in un'altra macchina

$ `touch ⟨new_file⟩`

- crea un nuovo file

$ `mkdir ⟨new_folder⟩`

- crea una nuova cartella

$ `cat ⟨file⟩`

- stampa sul terminale il contenuto del file

$ `> ⟨file⟩`

- stampa l'output sul file specificato

$ `ln -s ⟨path⟩/⟨file_or_directory⟩ ⟨path⟩/⟨file_or_directory⟩`

- crea un collegamento del primo oggetto nel luogo specificato dal secondo oggetto

$ `locate ⟨file⟩`

- elenca i file che presentano la stringa specificata nel loro nome. La ricerca viene fatta in tutta la macchina

$ `cat ⟨file⟩`

- stampa sul terminale il contenuto del file

$ `less ⟨file⟩`

- scorre sul terminale il contenuto del file

$ `head [-n ⟨number⟩] ⟨file⟩`

- stampa sul terminale le prime righe del file

$ `tail [-n ⟨number⟩] ⟨file⟩`

- stampa sul terminale le ultime righe del file

4.3 Processi

$ → `Ctrl+S`
- mette in pausa il processo in corso

$ → ⟨`any_key`⟩
- fa riprendere il processo messo in pausa

$ `jobs`
- stampa i processi in corso e la loro etichetta numerica

$ → `Ctrl+Z`
- addormenta il processo in corso

$ `fg` ⟨`number`⟩
- risveglia il processo con l'etichetta numerica specificata

$ `fg`
- risveglia tutti i processi addormentati

$ → `Ctrl+C`
- termina il processo in corso

$ ⟨`command`⟩
- avvia il processo e blocca il terminale

$ ⟨`command`⟩ `&`
- avvia il processo senza bloccare il terminale

$ → `Ctrl+Z` → `bg`
- se il terminale e' bloccata da un processo in corso, sblocca il terminale

$ `help` ⟨`command`⟩
- solitamente, stampa le istruzioni relative al comando

$ ⟨`command`⟩ `help`
- solitamente, stampa le istruzioni relative al comando

$ `kill` ⟨`pid_number`⟩
- termina il processo associato al numero PID specificato

$ ⟨`executable`⟩ `[-j`⟨`number`⟩`]`
- esegue il processo usando il numero di processori specificato. Di solito viene specificato `-j8`

$ `make`
- compila

$ `make clean`
- rimuove i file di compilazione. Prima di eseguire una nuova compilazione e' consigliato rimuovere i file di compilazione gia' esistenti, che potrebbero essere corrotti

$ ldd ⟨executable⟩

- elenca le librerie linkate dall'eseguibile specificato

$ watch ⟨executable⟩

- esegue il processo specificato ogni due secondi. In altre parole, esegue il processo specificato e lo aggiorna continuamente. ESEMPIO: `$ watch head output_file.txt`

$ runlim -s ⟨number⟩ ⟨command⟩

- esegue il comando specificato limitando i megabyte di RAM che il processo puo' usare al numero specificato

4.4 Scrittura

$ → Ctrl+Shift+C

- copia

$ → Ctrl+Shift+V

- incolla

$ clear

- pulisce il terminale

4.5 Grep

$ grep -Rn ‘‘⟨term⟩’’ ⟨folder⟩

- cerca il termine in tutti i file della cartella e delle sue sottocartelle, ricorsivamente

4.6 Percorsi

./

- la cartella attuale

⟨object⟩

- se non e' specificato un percorso, il percorso e' impostato di default nella cartella attuale

../

- la cartella superiore

./⟨folder⟩

- la sottocartella specificata

~

- la cartella HOME dell'utente attuale

/

- la cartella ROOT

./[⟨path⟩]

■ il percorso relativo

`/[⟨path⟩]`
■ il percorso assoluto

`/tmp`
■ la cartella dei file temporanei

4.7 Accesso da Remoto

$ `ssh ⟨machine⟩[.fbk.eu]`
■ accede al terminale macchina dall'interno del centro FBK come utente attuale

$ `ssh -X ⟨machine⟩[.fbk.eu]`
■ accede al terminale e all'interfaccia grafica della macchina dall'interno del centro FBK come utente attuale

$ `ssh ⟨user⟩%⟨machine⟩@gate.fbk.eu`
■ accede al terminale macchina fuori dal centro FBK

$ `ssh -X ⟨user⟩%⟨machine⟩@gate.fbk.eu`
■ accede al terminale e all'interfaccia grafica della macchina fuori dal centro FBK

$ `hostname`
■ stampa il nome della macchina attuale

$ → `Ctrl+D`
■ esce dalla macchina attuale

4.8 Script eseguibili

`#!⟨path⟩/⟨program⟩`
■ da incollare all'inizio del file per specificare che il file deve essere eseguito dal programma specificato

$ `chmod ⟨number⟩⟨number⟩⟨number⟩ ⟨file⟩`
■ rende eseguibile il file di script – il significato dei numeri e' piu' in basso

$ `./⟨file⟩`
■ esegue il file di script

$ `⟨file.sh⟩`
■ esegue il file di script, se il file e' nella cartella `/home/sbicego/script` o in un'altra cartella il cui path e' stato aggiunto a quelli ispezionati da BASH

4.8.1 Permessi

■ I permessi di un file eseguibile sono definiti da tre cifre.

- La posizione delle cifre ha il seguente significato:

  (1) i permessi del proprietario

  (2) i permessi degli utenti interni alla classe del file

  (3) i permessi degli utenti esterni alla classe del file

- Il valore delle cifre ha il seguente significato:

  (0) nessun permesso

  (1) permesso di esecuzione

  (2) permesso di scrittura

  (4) permesso di lettura

  (n) l'insieme dei permessi corrispondenti all'unica combinazione di cifre la cui somma coincide con la cifra **n**

- Il permesso consigliato e' `chmod 644` ⟨`file`⟩ per tutti i file e `chmod 744` ⟨`file`⟩ per gli eseguibili.

4.9 Sintassi

$ ‘⟨`command`⟩‘
- restituisce il valore di ritorno del comando

$ `$`⟨`variable`⟩
- restituisce il valore della variabile

$ ⟨`variable`⟩=⟨`value`⟩
- assegna il valore alla variabile (ed eventualmente la dichiara)

$ `unset` ⟨`variable`⟩
- elimina la variabile (solo nell'istanza attuale del terminale)

$ ⟨`instruction`⟩;⟨`instruction`⟩
- esegue in sequenza le istruzioni

4.10 Varie

$ `wall ‘‘`⟨`string`⟩`’’`
- stampa la stringa specificata in tutti i terminali della macchina corrente

5 EMACS

Descrizione: EMACS e' un editor di testo.

5.1 Scrittura

→ `Ctrl+X` → `S` → `Y`

■ salva

→ `Alt+W`

■ copia

→ `Ctrl+W`

■ taglia

→ `Ctrl+Y`

■ incolla

→ `Ctrl+X` → `U`

■ annulla

→ `Ctrl+X` → `H`

■ seleziona tutto

→ `Ctrl+S` → ⟨`term`⟩

■ cerca le occorrenze del termine specificato dal punto in cui si trova il cursore in avanti – una volta specificato un termine, la ripetizione del comando `Ctrl+S` scorre le occorrenze del termine

→ `Ctrl+R` → ⟨`term`⟩

■ cerca le occorrenze del termine specificato dal punto in cui si trova il cursore all'indietro – una volta specificato un termine, la ripetizione del comando `Ctrl+R` scorre le occorrenze del termine

→ `Alt+G+G` → ⟨`number`⟩

■ salta alla riga specificata

→ `AltGr+/`

■ scorre i termini candidati a completare la parola

→ `Shift+F1`

■ commenta la regione selezionata

→ `Shift+F2`

■ decommenta la regione selezionata

→ `Ctrl+Alt+F`

■ assunto di avere il cursore posizionato su una parentesi aperta, sposta il cursore sulla parentesi chiusa associata

→ `Ctrl+Alt+N`

■ assunto di avere il cursore posizionato su una parentesi aperta, sposta il cursore sulla parentesi chiusa associata – `N` sta per NEXT

→ `Ctrl+Alt+P`

■ assunto di avere il cursore posizionato su una parentesi chiusa, sposta il cursore sulla parentesi aperta associata – `P` sta per PREVIOUS

→ `Ctrl+X` → `(`
■ apre la memorizzazione di una macro da tastiera

→ `Ctrl+X` → `)`
■ chiude la memorizzazione della macro da tastiera

→ `Ctrl+X` → `E`
■ applica la macro da tastiera memorizzata – successivamente per continaure ad applicare la macro basta premere solo `E`

→ `name-last-kbd-macro` → ⟨`name`⟩
■ associa la macro da tastiera attualmente memorizzata al nome specificato – successivamente per applicare la macro basta scrivere il nome corrispondente nel menu' META-X

→ `insert-kbd-macro` → ⟨`name`⟩
■ salva permanentemente la macro specificata nel file di inizializzazione `.emacs` di EMACS

→ `Ctrl+X+C`
■ chiude il file attualmente aperto

→ `Ctrl+SpaceBar`
■ inserisce un MARK POINT nella posizione attuale del cursore. Questo comando equivale a posizionare il cursore nel punto specificato e tenere premuto il tasto `Shift`. In altre parole, invece di tenere premuti `Shift+Ctrl` basta premere una sola volta `Ctrl+SpaceBar`

→ `Alt+G` → `N`
■ muove il cursore alla segnalazione successiva del flycheck

→ `Alt+G` → `P`
■ muove il cursore alla segnalazione precedente del flycheck

→ `Ctrl+X Ctrl+F`
■ apre un nuovo file in un nuovo buffer nella finestra corrente

→ `Ctrl+D` → ⟨`number`⟩ ⟨`command`⟩
■ esegue il numero di volte specificato il comando specificato. Il comando deve essere singolo, non sono ammesse sequenze di piu' di un comando

→ `Ctrl+C` → `>`
■ aggiunge una tabulazione ad ogni riga della regione selezionata

→ `Ctrl+C` → `<`
■ toglie una tabulazione ad ogni riga della regione selezionata

5.1.1 Bookmark

Descrizione: Un BOOKMARK e' un collegamento ad un percorso.

→ `Ctrl+X` → `r` → `m` → ⟨`name`⟩

- salva il percorso attuale in un bookmark

→ `Ctrl+X` → `r` → `b` → ⟨`name`⟩

- salta al percorso del bookmark specificato

5.2 Finestre e Buffer

Spiegazione: Una finestra di EMACS contiene vari BUFFER. Una finestra puo' essere divisa in sottofinestre, e ciascuna sottofinestra contiene un BUFFER della finestra. Le sottofinestre e i BUFFER possono essere scorsi.

→ `Ctrl+X` → `3`

- divide la finestra attuale in due sottofinestre verticali

→ `Ctrl+X` → `2`

- divide la finestra attuale in due sottofinestre verticali

→ `Ctrl+X` → `1`

- rimuove tutte le sottofinestre a parte quella selezionata

→ `Ctrl+X` → `o`

- scorre le sottofinestre della finestra attuale

→ `Crtl+X` → ⟨`LEFT` or `RIGHT`⟩

- scorre i buffer della finestra attuale

→ `Crtl+X` → `Ctrl+F` → ⟨`file`⟩

- aggiunge il file specificato ai buffer della finestra attuale

→ `Crtl+X` → `K`

- termina il BUFFER selezionato

→ `Crtl+X` → `B`

- mostra tutti i buffer della finestra corrente e permette di selezionarli col mouse

→ `Ctrl+X` → `revert-buffer`

- aggiorna il buffer all'ultima versione. Se per esempio il file associato al buffer viene modificato da cause esterne, il comando aggiorna il buffer alla versione attuale del file

5.3 Terminale

→ `Alt+X` → `shell`

- apre il terminale all'interno della scheda attuale

5.4 Meta-X

Descrizione: il menu' META-X (scritto `M-x`) viene aperto cliccando `Alt+X` e viene chiuso all'esecuzione dell'istruzione.

→ `Alt+X` → `query-replace` → ⟨`term`⟩ → ⟨`new_term`⟩ → `[y] [n] [Shift+1]`

- sostituisce tutte le occorrenze del primo termine col secondo termine all'interno della regione selezionata. Se non e' selezionata nessuna regione, dal punto selezionato dal cursore in poi. Non sono ammessi caratteri speciali. `y` esegue la sostituzione dell'attuale occorrenza, `n` salta la sostituzione dell'attuale occorrenza, `Shift+1` esegue la sostituzione in tutte le occorrenze rimanenti

→ `Alt+X` → `query-replace-regexp` → ⟨`term`⟩ → ⟨`new_term`⟩ → `[y] [n] [Shift+1]`

- sostituisce tutte le occorrenze del primo termine col secondo termine all'interno della regione selezionata. Se non e' selezionata nessuna regione, dal punto selezionato dal cursore in poi. Sono ammessi caratteri speciali. `y` esegue la sostituzione dell'attuale occorrenza, `n` salta la sostituzione dell'attuale occorrenza, `Shift+1` esegue la sostituzione in tutte le occorrenze rimanenti

→ `ESC` → `ESC` → `ESC`

- esce dal menu META-X

→ `Alt+X` → `cc`

- toglie l'evidenziatore dalle parole

→ `Alt+X` → `highlight-regexp` → ⟨`regexp`⟩ → ⟨`color`⟩

- evidenzia col colore specificato tutte le occorrenze dell'espressione specificata

5.4.1 Scorciatoie

Descrizione: le scorciatoie del menu' META-X sono definite nel file `.emacs`.

5.4.2 Caratteri Speciali

`^`

- inizio riga

`$`

- fine riga

`.`

- qualsiasi carattere a parte INIZIO RIGA `^` e FINE RIGA `$`

`a-z`

- una lettera minuscola

`A-Z`

- una lettera maiuscola

`0-9`

- una cifra

`[`⟨`characters_set`⟩`]`

- le parentesi quadre racchiudono un insieme di caratteri

`[^⟨characters_set⟩]`
- le parentesi quadrate con cappuccio racchiudono l'insieme dei caratteri esclusi

`*`
- almeno 0 volte il carattere immediatamente precedente

`+`
- almeno 1 volta il carattere immediatamente precedente

`\|`
- disgiunzione logica

`\(⟨characters_set⟩\)`
- etichetta il contenuto delle parentesi con un numero

`\⟨number⟩`
- stampa il contenuto etichettato col numero specificato

`→ Ctrl+Q → Ctrl+J`
- a capo

5.5 Lisp

Spiegazione: EMACS viene personalizzato attraverso il linguaggio LISP. Elenchiamo alcuni comandi legati al linguaggio LISP.

`→ Alt+X → list-colors-display`
- elenca tutti i colori che si possono usare per colorare i testi di EMACS

5.6 Modi

Definizione: Un MODO e' un insieme di funzionalita' che vengono applicate a uno o piu' buffer di EMACS.

Modo maggiore: Ad un buffer di EMACS puo' essere associato al piu' un solo MODO MAGGIORE. Per quello che ci riguarda il modo maggiore consiste nella colorazione e nell'indentazione del testo di un buffer.

Modo minore: Ad un MODO MAGGIORE possono essere associati piu' MODI MINORI. Per quello che ci riguarda il modo minore consiste nella segnalazione degli errori sintattici del testo di un buffer.

5.6.1 Modo maggiore: colorazione e indentazione

Solitamente il modo maggiore viene definito in un file con estensione `.el` che va posizionato nella cartella MODI di EMACS. All'interno del file di inizializzazione `.emacs` di EMACS viene associato il modo maggiore al file `.el` e si specificano quali tipi di file devono essere associati al modo maggiore.

5.6.2 Modo minore: segnalazione degli errori

Elenchiamo i passi per definire il modo minore che implementa la segnalazione degli errori.

- Si implementa un eseguibile, chiamiamolo `parse.py`, che prende in input il nome di un file di testo (quello che successivamente sara' il nome del buffer di EMACS) e restituisce (stampato sullo standard output) un elenco di stringhe che rappresentano gli errori presenti nel file di testo.

  Solitamente si ha gia' a disposizione un eseguibile, chiamiamolo `exe`, di cui ci interessa vedere segnalata la sintassi direttamente sui file sorgenti che verranno dati in input a tale eseguibile. L'eseguibile `parse.py` quindi consiste nell'applicazione di `exe` al file sorgente e alla stampa su standard output degli errori segnalati dal parser o dai checker gia' implementati in `exe`.

  Il formato delle stringhe in output dipende da che tipo di FLYCHECKER si vuole usare (*vedi punto successivo*).

- Nel file di inizializzazione `.emacs` di EMACS si definisce un modo minore di tipo FLYCHECKER e lo si associa all'eseguibile `parse.py`.

  Sempre in `.emacs` si associa il modo minore FLYCHECKER al modo maggiore o ai tipi di file desiderati.

Quello che succede all'apertura di un buffer di EMACS associato al modo minore corrispondente alla segnalazione degli errori e' questo:

- EMACS apre il buffer.
- A ciclo continuo vengono eseguite le seguenti operazioni.
  - Il testo del buffer viene ricopiato in un altro file temporaneo.
  - Il file temporaneo viene dato in input a `parse.py`.
  - Il file temporaneo viene dato in input a `exe`.
  - `exe` stampa i messaggi d'errore nel proprio standard error.
  - `parse.py` legge i messaggi d'errore, eventualmente li manipola, e li stampa nel proprio standard output.
  - EMACS riceve questi messaggi d'errore, li parsa e li segnala nel buffer.

6 C

6.1 CMake

Descrizione: CMAKE definisce un progetto C e genera le istruzioni per la sua compilazione.

Comandi:

project>build $ `cmake ..`
- crea gli script CMAKE che guidano la compilazione

project>build>CMakeChache.txt → `CMAKE_BUILD_TYPE:STRING=Debug`
- abilita il debugger

project>build>CMakeChache.txt → `CMAKE_BUILD_TYPE:STRING=Release`
- ottimizza il tempo d'esecuzione

project>build>CMakeChache.txt → `CMAKE_BUILD_TYPE:STRING=MinSizRel`
- ottimizza le dimensioni dell'eseguibile

project>build $ `make`
- compila il progetto

Requisiti:

- il programma CMake, di versione 2.8 o superiore
- il compilatore gcc

6.2 Progetto

Descrizione: un progetto c e' definito dalle seguenti cartelle:

- `project`
- `project>src`
- `project>build`
- `project>docs`
- `project>examples`

e dai seguenti file:

- `project>CMakeList.txt`
- `project>src>CMakeList.txt`
- `project>src>main.c`
- `project>src>library.c`
- `project>src>library.h`

6.2.1 Allegati

- la cartella `project` contiene il prototipo di un progetto c.

6.3 Notazione

Variabili: lettere minuscole e underscore
Funzioni: lettere minuscole e underscore
Domini: lettere iniziali maiuscole, senza underscore

6.4 Valgrind

Descrizione: Valgrid e' un debugger di progetti c che esegue in modo interattivo l'eseguibile. In particolare, individua le istruzioni in cui si accede ad aree di memoria non allocata e verifica se ogni `malloc` viene corrisposta alla relativa `free`.

Comandi:

project>src>⟨file.c⟩ → `Alt+X` → `gud-gdb`
- abilita l'esecuzione controllata dell'eseguibile generato dal file

project>build $ `valgrind --leak-check=full ./`⟨`executable`⟩

- avvia l'esecuzione controllata dell'eseguibile

Requisiti:

- il programma GDB, che solitamente e' integrato in GCC

6.5 GTags

Descrizione: GTAGS genera la genealogia della variabili e delle funzioni di un progetto C e individua il punto di codice dove e' stato definito ciascun oggetto.

Comandi:

project $ `gtags`
- genera la genealogia del progetto dalla cartella corrente in giu' – ha senso chiamare questo comando dalla cartella superiore del progetto

$ `global` ⟨`object_name`⟩
- stampa il nome del file dove e' definito l'oggetto specificato – questo comando e' sconsigliato perche' scomodo

⟨file.c⟩ → `Alt+X` → `xgtags-mode`
- abilita la modalita' GTAGS

⟨file.c⟩ → `Alt+.` → ⟨`term`⟩
- salta al punto in cui viene definito il termine – il termine indicato di default e' quello selezionato dal cursore del mouse; per selezionare un altro termine scriverlo nel menu' `M-x`; se esistono piu' definizioni del termine, il salto avviene su quella correlata al codice da cui parte il salto; se si vuole saltare ad un'altra delle definizioni, cliccare su quella desiderata e premere INVIO

⟨file.c⟩ → `Alt+,`
- torna al punto precedente all'ultimo salto

6.6 GDB

Descrizione: GDB associa il file sorgente di un eseguibile C alla sua esecuzione. L'esecuzione viene svolta in modo interattivo.

Avvio: le seguenti istruzioni sono valide se il progetto C e' definito con CMAKE.

- Eseguire con la SHELL il comando `project/build $ cmake ..`
- Nel file `project/build/CMakeCache.txt` aggiungere il termine `Debug` in modo che una delle righe sia `CMAKE_BUILD_TYPE:STRING=Debug`
- Eseguire con la SHELL il comando `project/build $ make`
- Aprire EMACS
- Aprire il menu `M-x` ed eseguire il comando `gdb [-i=mi` ⟨`path`⟩⟨`executable`⟩`]` – facoltativamente, si puo' specificare gia' ora l'eseguibile
- Nel menu `(gdb)` eseguire il comando `file [`⟨`path`⟩`]`⟨`executable`⟩ – il comando va eseguito solo se l'eseguibile non e' gia' stato specificato
- Prima di eseguire il programma col comando `run` aggiungere almeno un BREAKPOINT, altrimenti il programma termina senza dare la possibilita' di fare l'analisi interattiva.

Comandi:

(gdb) Ctrl+UP
- riscrive nel menu l'ultimo comando precedentemente eseguito

(gdb) ENTER
- esegue l'ultimo comando precedentemente eseguito

(gdb) `run` ⟨`command_line_arguments`⟩
- esegue il file eseguibile con gli argomenti da terminale specificati

(gdb) `kill`
- interrompe l'esecuzione

(gdb) `quit`
- esce da GDB

(gdb) `break` ⟨`file.c`⟩:⟨`line_number`⟩
- inserisce un BREAKPOINT alla prima occorrenza dell'esecuzione del punto del codice sorgente specificato

(gdb) `break` ⟨`file.c`⟩:⟨`function`⟩
- inserisce un BREAKPOINT alla prima chiamata della funzione specificata

→ `Ctrl+X` → `SPACEBAR`
- inserisce un BREAKPOINT nella riga selezionata

(gdb) `continue`
- continua l'esecuzione fino al prossimo BREAKPOINT o al prossimo punto di interazione con l'utente

(gdb) `clear`
- rimuove il BREAKPOINT della riga selezionata

(gdb) `delete`
- rimuove tutti i BREAKPOINT

(gdb) `step` [⟨`number`⟩]
- esegue l'istruzione successiva, o il numero di istruzioni successive se specificato. L'esecuzione interattiva non entra nel corpo delle funzioni chiamate: le funzioni chiamate vengono eseguite completamente e l'esecuzione si ferma alla riga successiva a quella della funzione chiamata

(gdb) `next` [⟨`number`⟩]
- esegue l'istruzione successiva, o il numero di istruzioni successive se specificato. L'esecuzione entra nel corpo delle funzioni chiamate

(gdb) `until` [⟨`file.c/line_number`⟩ or ⟨`file.c/function`⟩]
- esegue il programma fino alla prima occorrenza dell'esecuzione dell'oggetto speficicato

(gdb) `where`
- stampa il numero di riga e la funzione in esecuzione dello stato attuale

(gdb) `jump` ⟨`line_number`⟩
- esegue il programma fino alla riga specificata all'interno del file attuale

(gdb) `backtrace`
- stampa lo stack nello stato attuale

(gdb) `up [⟨number⟩]`
- sale di una posizione nello stack, o del numero di posizioni specificato

(gdb) `down [⟨number⟩]`
- scende di una posizione nello stack, o del numero di posizioni specificato

(gdb) `print ⟨variable⟩`
- stampa il valore della variabile specificata nello stato attuale

(gdb) `delete breakpoints`
- cancella i breakpoint attuali

6.6.1 Scorciatoie

(gdb) `bt`
- stampa lo stack

(gdb) `up`
- sale nello stack

(gdb) `down`
- scende nello stack

(gdb) `n`
- esegue l'istruzione successiva. Se viene chiamata una funzione, l'esecuzione entra nel corpo della funzione

(gdb) `s`
- esegue l'istruzione successiva. Se viene chiamata una funzione, l'intera funzione viene eseguita

(gdb) `c`
- continua l'esecuzione fino al prossimo breakpoint o fino alla prossima interruzione del'esecuzione

(gdb) `r`
- avvia l'esecuzione del programma

(gdb) `d breakpoints`
- cancella i breakpoint attuali

7 CPP

Descrizione: il preprocessore CPP definisce che parti di codice sorgente devono essere compilate e sposta pezzi di codice sorgente secondo le istruzioni che gli vengono fornite.

Istruzioni:

`#include ‘‘[⟨path⟩/]⟨file⟩’’`
- incolla il contenuto del file specificato nel file attuale, al posto dell'istruzione `#include` stessa

`#define ⟨term⟩ ⟨value⟩`
- ogni occorrenza del termine viene sostituita dal valore

`#define ⟨term⟩`
- il CPP informa se stesso che il termine e' definito

`#ifdef ⟨term⟩`
- il codice successivo alla direttiva viene compilato solo se il termine e' gia' stato definito

`#ifndef ⟨term⟩`
- il codice successivo alla direttiva viene compilato solo se il termine non e' gia' stato definito

`#if ⟨boolean_term⟩`
- il codice successivo alla direttiva viene compilato solo se il termine booleano e' vero

`#else`
- il codice successivo alla direttiva viene compilato solo se la condizione dell'`#ifdef`, dell'`#ifndef` o dell'`#if` immediatamente precedente non e' soddisfatta

`#endif`
- chiude il contesto aperto dall'`#ifdef`, dall'`#ifndef` o dall'`#if` immediatamente precedente

Comandi:

$ `cpp ⟨input_file⟩ ⟨output_file⟩`
- esegue le direttive del file di input e stampa il risultato nel file di output

Requisiti:

- il preprocessore CPP, solitamente integrato in GCC

8 Latex

Descrizione: LATEX e' un compilatore che produce file di testo e file grafici.

8.1 Compilazione

$ `latex ⟨file.tex⟩`
- compila il file `.tex` in `.dvi`

$ `pdflatex ⟨file.tex⟩`
- compila il file `.tex` in `.pdf`

$ `latexmk -pdf -pvc ⟨file.tex⟩`
- compila il file `.tex` in `.pdf` automaticamente ad ogni salvataggio

8.2 Documento

```
\documentclass[a4paper,12pt,fleqn,leqno,twoside,openany]{article}
```

- il documento di tipo ARTICOLO

```
\documentclass[final,pdftex,a4paper,12pt,leqno]{beamer}
```
- il documento di tipo SLIDE

```
\mode<presentation>{\usetheme{Madrid}}
\beamertemplatenavigationsymbolsempty
```
- lo stile del documento di tipo SLIDE

8.3 Librerie

```
\usepackage[utf8]{inputenc}
\usepackage[english]{babel}
\usepackage{latexsym}
\usepackage{amsmath}
\usepackage{amssymb}
\usepackage{amsfonts}
\usepackage{mathrsfs}
\usepackage{makeidx}
\usepackage{amsmath}
\usepackage{graphicx}
\usepackage{makeidx}
\usepackage{booktabs}
\usepackage{geometry}
\usepackage{url}
\usepackage{xspace}
\usepackage{xcolor}
\usepackage{verbatim}
\usepackage{listings}
\usepackage{color}
```
- le librerie

8.4 Comandi Personalizzati

```
\newcommand{\n}{\textnormal} % normal
\renewcommand{\i}{\textit} % italic
\renewcommand{\b}{\textbf} % bold
\newcommand{\tw}{\texttt} % typewriter
\renewcommand{\sc}{\textsc} % small caps
\renewcommand{\cal}[1]{$\mathcal{#1}$} % calligraphic
\newcommand{\mat}[1]{$\mathcbb{#1}$} % mathematical
\newcommand{\pn}[1]{\n{#1} } % proper name
```
- gli stili di testo

```
\newcommand{\bbox}{{\scriptsize $\blacksquare$}} % black box
\newcommand{\bball}{{\scriptsize $\bullet$}} % black circle
\newcommand{\skipline}{\vspace{\baselineskip}} % skip line
```
- i comandi personalizzati

```
\geometry{left=20mm, right=20mm, top=15mm, bottom=15mm}
```
- i bordi della pagina

8.5 Comandi Standard

```
\.  \, \:  \; \quad \qquad
\hspace{⟨number⟩[pt][cm]}
```

- le spaziature orizzontali

```
\smallskip \medskip \bigskip
\baselineskip
\vspace{⟨number⟩[pt][cm]}
```

- le spaziature verticali

```
\bigr \Bigr \biggr \Biggr
\bigl \Bigl \biggl \Biggl
```

- le parentesi

```
\displaystyle
```

- il formato delle formule

```
\cite{⟨term⟩}
\cite[⟨name⟩]{⟨term⟩}
```

- le citazioni

```
-
--
---
```

- i trattini

8.6 Indentazione

```
\setlength{\parindent}{0pt}
```

- l'indentazione

```
\addtolength{\itemindent}{1cm}
```

- l'indentazione all'interno dell'ambiente `itemize`

8.7 Codice

```
\definecolor{white}{rgb}{1,1,1}
\definecolor{yellow}{rgb}{1,0.8,0}
\definecolor{light_blue}{rgb0.12,0.56,1}
\definecolor{orange}{rgb}{1,0.6,0}
\definecolor{red}{rgb}{0.78,0.03,0.08}
\definecolor{black}{rgb}{0,0,0}
\definecolor{green}{rgb}{0.01,0.75,0.24}
```

- i colori dell'ambiente `lstlisting`

```
\lstset{
language=[ANSI]C,
backgroundcolor=\color{bianco},
basicstyle=\color{nero}\normalfont\small,
identifierstyle=\color{nero},
```

```
keywords=[1]{int},
keywordstyle=[1]\color{verde}\textbf,
keywords=[2]{if,then,else,for,while,do,return},
keywordstyle=[2]\color{azzurro}\textbf,
commentstyle=\color{rosso},
stringstyle=\color{rosso},
xleftmargin=-1.8cm,
xrightmargin=-1.8cm,
rulecolor=\color{nero},
frame=none,
breaklines=true,
captionpos=b,
keepspaces=true,
showstringspaces=false,
tabsize=4,
extendedchars=true
}
```

- lo stile dell'ambiente `lstlisting` per il codice C

8.8 Dati

```
\chapter{⟨name⟩}
\section{⟨name⟩}
\subsection{⟨name⟩}
\subsubsection{⟨name⟩}
\tableofcontents
```

- l'indice

```
\addcontentsline{toc}{section}{Bibliography}
\begin{thebibliography}{}
\bibitem{⟨reference⟩} ⟨authors⟩:  \s{⟨title⟩}.  ⟨editer⟩, ⟨date⟩.
\end{thebibliography}
```

- la bibliografia

```
\title[⟨short_title⟩]{⟨title⟩}
\author[⟨short_authors⟩]{⟨authors⟩}
\institute[⟨short_institute⟩]{⟨institute⟩}
\date{⟨date⟩}
\titlepage
```

- la titolazione

8.9 Allegati

- la cartella `latex_docs` contiene i prototipi di alcuni documenti LATEX.

9 nuXmv

9.1 Lettura

```
$ nuXmv -int
```

- avvia nuXmv in modalita' interattiva

nuXmv $ `set pp_list cpp`
- abilita l'uso del preprocessore CPP

$ `nuXmv -int -pre cpp`
- avvia nuXmv col preprocessore CPP abilitato

nuXmv $ `read_model -i ⟨file.smv⟩`
- legge il modello contenuto nel file

nuXmv $ `reset`
- chiude il modello attualmente letto

9.2 Compilazione

9.2.1 Modelli Finiti

nuXmv $ `go`
- compila il modello

nuXmv $ `check_ltlspec [-k <⟨number⟩>]`
- verifica le specifiche di tipo LTLSPEC, limitando il numero di stati controllati al valore specificato

nuXmv $ `check_ltlspec -p ‘‘⟨formula⟩’’ [-k <⟨number⟩>]`
- verifica la specifica di tipo LTLSPEC indicata, limitando il numero di stati controllati al valore specificato

nuXmv $ `check_invar [-k <⟨number⟩>]`
- verifica le specifiche di tipo INVARSPEC, limitando il numero di stati controllati al valore specificato

nuXmv $ `check_invar -p ‘‘⟨formula⟩’’ [-k <⟨number⟩>]`
- verifica la specifica di tipo INVARSPEC indicata, limitando il numero di stati controllati al valore specificato

nuXmv $ `check_ctlspec [-k <⟨number⟩>]`
- verifica le specifiche di tipo CTLSPEC, limitando il numero di stati controllati al valore specificato

nuXmv $ `check_ctlspec -p ‘‘⟨formula⟩’’ [-k <⟨number⟩>]`
- verifica la specifica di tipo CTLSPEC indicata, limitando il numero di stati controllati al valore specificato

nuXmv $ `pick_state -i`
- fa scegliere lo stato iniziale

nuXmv $ `simulate -i`
- avvia la simulazione interattiva del modello

9.2.2 Modelli Infiniti

nuXmv $ `go_msat`
- compila il modello

nuXmv $ `msat_check_ltlspec_bmc [-k <⟨number⟩>]`

- verifica le specifiche di tipo LTLSPEC, limitando il numero di stati controllati al valore specificato

nuXmv $ `msat_check_ltlspec_bmc -p ‘‘⟨formula⟩’’ [-k <⟨number⟩>]`

- verifica la specifica di tipo LTLSPEC indicata, limitando il numero di stati controllati al valore specificato

nuXmv $ `msat_check_invar_bmc [-k <⟨number⟩>]`

- verifica le specifiche di tipo INVARSPEC, limitando il numero di stati controllati al valore specificato

nuXmv $ `msat_check_invar_bmc -p ‘‘⟨formula⟩’’ [-k <⟨number⟩>]`

- verifica la specifica di tipo INVARSPEC indicata, limitando il numero di stati controllati al valore specificato

nuXmv $ `msat_check_ctlspec_bmc [-k <⟨number⟩>]`

- verifica le specifiche di tipo CTLSPEC, limitando il numero di stati controllati al valore specificato

nuXmv $ `msat_check_ctlspec_bmc -p ‘‘⟨formula⟩’’ [-k <⟨number⟩>]`

- verifica la specifica di tipo CTLSPEC indicata, limitando il numero di stati controllati al valore specificato

9.3 Analisi

nuXmv $ `write_flat_model -o ⟨file.smv⟩`

- stampa sul file specificato l'appiattimento del modello attualmente in lettura

9.4 Scorciatoie

nuXmv $ `-source ⟨nuXmv_shell_file⟩`

- avvia NUXMV ed esegue le istruzioni da shell interattiva contenute nel file specificato

10 Repository

Descrizione: una repository e' un server che permette il salvataggio, l'accesso e la modifica di file e cartelle e che conserva la storia delle modifiche che i file e le cartelle hanno subito. Una repository consiste in un insieme di dati sul server e in una loro copia locale presente in una cartella della macchina. Ciascun dato ha due versioni: una locale e una sul server. L'utente modifica i dati locali e poi impone ai dati sul server di aggiornarsi alle loro versioni locali, oppure impone ai dati locali di aggiornarsi alle loro versioni sul server.

10.1 SVN

⟨folder⟩ $ `svn co ⟨web_address⟩`

- la repository dell'indirizzo specificato viene clonata nella cartella attuale della macchina

⟨svn_folder⟩ $ `svn status`

- stampa le differenze tra la repository nel server e il suo clone locale. `?` vuol dire che il file e' presente nella cartella ma non nel server, `A` vuol dire che il file e' presente nella cartella e non nel server ma che e'

stato regolarmente aggiunto al server, `M` vuol dire che il file e' presente sia nella cartella che nel server ma che le due versioni sono diverse, `D` vuol dire che il file e' stato regolarmente rimosso sia dalla cartella sia dal server, `!` vuol dire che il file e' presente nel server ma non nella cartella

⟨svn_folder⟩ $ `svn add ⟨file⟩`

■ informa il programma di voler copiare il file dalla cartella e di volerlo incollare nel server – e' necessario che una versione del file non sia gia' presente nel server

⟨svn_folder⟩ $ `svn up ⟨file⟩`

■ informa il programma di voler copiare il file dalla cartella e di volerlo incollare nel server – e' necessario che una versione del file sia gia' presente nel server

⟨svn_folder⟩ $ `svn rm ⟨file⟩`

■ rimuove il file dalla cartella e informa il programma di voler rimuovere il file dal server

⟨svn_folder⟩ $ `svn commit`

■ modifica i file del server in base alle istruzioni specificate in precedenza

⟨svn_folder⟩ $ `svn revert ⟨file⟩`

■ copia il file dal server e lo incolla nella cartella, sostituendo l'eventuale versione locale

⟨svn_folder⟩ $ `svn up`

■ informa il programma di voler copiare i file dal server e di volerli incollare nella cartella, conservando le differenze delle versioni locali che non sono in contraddizione con le versioni nel server

⟨svn_folder⟩ $ `svn log [-v]`

■ stampa l'elenco dei COMMIT in ordine cronologico, coi rispettivi codici. Se l'opzione `-v` e' specificata vengono stampate le modifiche effettuate su ciascun file per ogni COMMIT

⟨svn_folder⟩ $ `svn checkout -r ⟨commit_code⟩`

■ copia la versione sul server associata al COMMIT indicato nella cartella locale

⟨svn_folder⟩ $ `svn rm --keep-local ⟨file⟩`

■ retrocede il file specificato da file addato a file non addato, senza pero' cancellarlo dal locale

10.1.1 Caratteri Speciali

*

■ tutte le stringhe

`[⟨string⟩]*[⟨string⟩]`

■ tutte le stringhe che cominciano e terminano con le successioni di caratteri indicate

10.2 GitLab

Descrizione: GITLAB permette di creare piu' rami di storia e di unire due rami mantenendo i dati di entrambi, a meno che non siano contraddittori. Permette anche di creare rami di storia locali.

⟨folder⟩ $ `git clone ⟨web_address⟩ [''⟨alias_name⟩'']`

■ la repository dell'indirizzo specificato viene clonata nella cartella attuale della macchina, eventualmente rinominando la cartella col nome specificato

⟨git_folder⟩ (⟨branch⟩) $ `git status`

■ stampa le differenze tra il ramo selezionato locale e la cartella

⟨git_folder⟩ (⟨branch⟩) $ `git diff [file_name]`

■ stampa su terminale le differenze, riga per riga, tra la versione del file nella cartella e la versione del file sul server (se il file non e' specificato, stampa le differenze tra tutti i file che contengono differenze)

⟨git_folder⟩ (⟨branch⟩) $ `git add ⟨file⟩`

■ informa il programma di voler copiare il file dalla cartella e di volerlo incollare nel ramo selezionato locale – e' necessario che una versione del file non sia gia' presente nel ramo selezionato del server

⟨git_folder⟩ (⟨branch⟩) $ `git up ⟨file⟩`

■ informa il programma di voler copiare il file dalla cartella e di volerlo incollare nel ramo selezionato locale – e' necessario che una versione del file sia gia' presente nel ramo selezionato locale

⟨git_folder⟩ (⟨branch⟩) $ `git rm ⟨file⟩`

■ rimuove il file dalla cartella e informa il programma di voler rimuovere il file dal ramo selezionato locale

⟨git_folder⟩ (⟨branch⟩) $ `git commit`

■ modifica i file del ramo selezionato locale in base alle istruzioni specificate in precedenza

⟨git_folder⟩ (⟨branch⟩) $ `git push origin ⟨branch⟩`

■ unisce il ramo locale selezionato al rispettivo ramo sul server, conservando i dati di entrambi, a meno che non siano contraddittori, e in tal caso chiede di specificare manualmente l'unione dei punti critici

⟨git_folder⟩ (⟨branch⟩) $ `git push`

■ unisce tutti i rami locali ai rispettivi rami sul server, conservando i dati di entrambi, a meno che non siano contraddittori, e in tal caso chiede di specificare manualmente l'unione dei punti critici

⟨git_folder⟩ (⟨branch⟩) $ `git pull`

■ copia i file dal ramo selezionato sul server e li incolla nella cartella

⟨git_folder⟩ (⟨branch⟩) $ `git checkout ⟨branch⟩`

■ passa al ramo indicato

⟨git_folder⟩ (⟨branch⟩) $ `git checkout -b ⟨branch⟩`

■ crea a partire dal punto del ramo selezionato un nuovo ramo locale che viene chiamato col nome indicato

⟨git_folder⟩ (⟨branch⟩) $ `git push origin ⟨branch⟩`

■ promuove il ramo locale selezionato a ramo sul server

⟨git_folder⟩ (⟨branch⟩) $ `git branch`

■ stampa i rami locali

⟨git_folder⟩ (⟨branch⟩) $ `git branch -r`

■ stampa i rami remoti

⟨git_folder⟩ (⟨branch⟩) $ `git branch -a`
- stampa tutti i rami, sia locali sia remoti

⟨git_folder⟩ (⟨branch⟩) $ `git reset --hard` ⟨`code`⟩
- torna al punto di ramo associato al commit di codice indicato

⟨git_folder⟩ (⟨branch⟩) $ `git branch -D` ⟨`branch`⟩
- il ramo indicato rimane sul server ma viene cancellato dal locale

⟨git_folder⟩ (⟨branch⟩) $ `git clean -f -d`
- elimina dal ramo locale i file, gli `add` e i `commit` che non sono stati pushati nel ramo sul server

⟨git_folder⟩ (⟨branch⟩) $ `git reset --hard origin/master`
- elimina dal ramo locale i file, gli `add` e i `commit` che non sono stati pushati nel ramo `master`

⟨git_folder⟩ (⟨branch⟩) $ `git fetch` → `git merge origin/master`
- copia i file del ramo `master` sul server e li incolla nel ramo locale selezionato

⟨git_folder⟩ (⟨branch⟩) $ `git format-patch` ⟨`branch`⟩ `--stdout` → ⟨`name`⟩`.patch`
- genera un file patch con le differenze tra il ramo attuale e il ramo indicato

⟨git_folder⟩ (⟨branch⟩) $ `gitk`
- apre l'interfaccia grafica – viene usato solo per la lettura

10.2.1 File

⟨git_folder⟩ (⟨branch⟩) $ `git checkout --` ⟨`folder` or `file`⟩
- l'oggetto specificato viene ripristinato localmente alla sua versione nel server

⟨git_folder⟩ (⟨branch⟩) $ `git reset HEAD` ⟨`file`⟩
- retrocede un file addato ad un file non addato

⟨git_folder⟩ (⟨branch⟩) $ `git log [--author=`⟨`name`⟩`] [-p` ⟨`path`⟩`]`
- stampa la storia dei commit dell'autore specificato riguardanti il percorso specificato. Se l'autore non viene specificato vengono mostrati tutti gli autori, e se il percorso non viene specificato viene considerata la cartella dell'intera repository

⟨git_folder⟩ (⟨branch⟩) $ `git log [--stat]`
- stampa la storia dei commit. Se l'opzione `--stat` e' specificata, per ogni commit vengono mostrati i file modificati

⟨git_folder⟩ (⟨branch⟩) $ `git checkout --` ⟨`file`⟩
- ripristina il file locale alla versione sul server

⟨git_folder⟩ (⟨branch⟩) $ `git add -e`
- stampa le differenze tra i file locali e quelli sul branch e fa scegliere riga per riga quali modifiche addare – questo comando e' consigliato

⟨git_folder⟩ (⟨branch⟩) $ `git add -p`

■ chiede in modo interattivo quali file o pezzi di file si vogliono addare. Il significato delle lettere stampate e' il seguente:

`y` : aggiungi il pezzo attuale

`n` : non aggiungere il pezzo attuale

`q` : non aggiungere i pezzi rimanenti

`s` : dividi ulteriormente il pezzo attuale

`e` : dividi manualmente il pezzo attuale

`?` : stampa le istruzioni

10.2.2 Branch

⟨git_folder⟩ (⟨branch⟩) $ `git reset --hard ⟨code⟩`
■ il puntatore di storia attuale passa al commit associato al codice specificato, ignorando ma mantenendo tutti i commit successivi

⟨git_folder⟩ (⟨branch⟩) $ `git rebase -p -onto ⟨code⟩^ ⟨code⟩`
■ il puntatore attuale di storia ignora il commit indicato, ma lo mantiene

⟨git_folder⟩ (⟨branch⟩) $ `git commit -c HEAD`
■ promuove il puntatore di storia attuale a testa del branch, cioe' rimuove localmente tutti i commit attualmente ignorati

⟨git_folder⟩ (⟨branch⟩) $ `git push origin -f`
■ esporta le modifiche locali sul server. In particolare, rimuove dal server i commit rimossi in locale

⟨git_folder⟩ (⟨branch⟩) $ `git reset HEAD^`
■ muove il puntatore al penultimo commit, senza modificare i file

⟨git_folder⟩ (⟨branch⟩) $ `git commit --amend`
■ aggiunge le modifiche all'ultimo commit invece di crearne uno nuovo

⟨git_folder⟩ (⟨branch⟩) $ `git push origin ⟨code⟩:⟨branch⟩`
■ pusha nel branch specificato tutti i commit fino a quello di codice specificato, non pushando ma mantenendo in locale i commit successivi

⟨git_folder⟩ (⟨branch⟩) $ `git revert ⟨code⟩`
■ applica un commit che cancella le modifiche del commit di codice specificato. Nella sostanza, cancella il commit indicato ma in modo costruttivo, cioe' aggiungendo un nuovo commit che esegue le azioni contrarie di quelle del commit che vuole essere cancellato

⟨git_folder⟩ (⟨branch⟩) $ `git rebase -i origin/⟨branch⟩`
■ apre una finestra interattiva in cui viene stampato l'elenco dei commit successivi all'ultimo push e viene permessa la loro modifica – viene usato per modificare o cancellare commit locali

⟨git_folder⟩ (⟨branch⟩) $ `git merge ⟨branch⟩`
■ mergia il branch specificato nel branch corrente

10.2.3 Merge

Descrizione: per richiedere l'aggiunta di un ramo locale ad un ramo sul server, eseguire le seguenti istruzioni:

- andare nella pagina web della repository GitLab: `https://gitlab.fbk.eu/`
- eventualmente cliccare sull'icona `Dashboard`, in alto a sinistra, dove compare la faccia di una volpe
- cliccare sul tasto `Create Merge Request`, il alto a destra
- eventualmente cliccare sul tasto `Change branches`, in alto a destra, e indicare quale ramo si vuole immergere in quale ramo
- compilare la scheda e cliccare sul tasto `Submit merge request`, in basso

o in alternativa, eseguire le seguenti istruzioni:

- digitare ⟨git_folder⟩ (⟨branch⟩) $ `git format-patch` ⟨`branch`⟩ `--stdout` → ⟨`name`⟩`.patch`
- osservare che e' stato generato un file di patch con le differenze tra il ramo attuale e il ramo indicato
- inviare il file di patch all'amministratore del ramo che vuole essere aggiornato
- l'amministratore sceglie se applicare la patch al ramo da aggiornare o meno

11 Python

11.1 Terminale

$ `python` ⟨`file.py`⟩
- esegue il file

⟨file.py⟩ $ → `Alt+X` → `python-checker` → [⟨`file.py`⟩]
- avvia il debugger

python-checker $ → `G`
- ricalcola gli errori del file

11.2 Sintassi

Allegato: La cartella `python_syntax` contiene un file che descrive i costrutti sintattici piu' comuni di PYTHON.

Descrizione: qui di seguito riportiamo alcuni tipi di oggetti.

MUTABILI
- vengono passati per riferimento

IMMUTABILI
- vengono passati per valore

ITERABILI
- possono incorrere in cicli FOR

CONTESTUALI
- possono incorrere in operatori WITH AS

Descrizione: qui di seguito riportiamo alcuni costrutti sintattici notevoli.

⟨string_1⟩+⟨string_1⟩
- concatena due stringhe che si trovano sulla stessa riga

⟨string_1⟩\
⟨string_2⟩
- concatena due stringhe che si trovano su righe diverse

⟨partial_instruction_1⟩\
⟨partial_instruction_2⟩
- il BACKSLASH permette di spezzare un'istruzione su piu' righe

11.3 Librerie

`from __future__ import absolute_import`
- obbliga i path degli `import` ad essere relativi ad una cartella di `sys.path`, cioe' vieta l'uso implicito della cartella corrente `os.getcwd()` come possibile cartella da cui specificare path di importazione (a meno che non sia in `sys.path`)

`__init__.py`
- durante l'esecuzione di un `import`, ogni volta che si passa attraverso una cartella (anche solo transitoriamente) viene eseguito il contenuto dell'eventuale file __init__.py contenuto in quella cartella

11.4 Importazione da C

Descrizione: in un file PYTHON e' possibile importare oggetti (solitamente funzioni) implementate e compilate in C. Elenchiamo i passi che bisogna seguire per creare una libreria C importabile in PYTHON.

L'esempio che riportiamo si riferisce ai file che si trovano in allegato nella cartella `python_c`.

- Prendiamo il file C con gli oggetti che ci interessano esportare in PYTHON. In allegato il file e' `file.c`.
- Creiamo un file con estensione `.i` in cui elenchiamo gli oggetti che vogliamo importare in PYTHON e diamo un nome alla libreria C che vogliamo creare. In allegato il file e' `file.i` e il nome della libreria e' `my_library`.

  La sintassi del file `.i` e' semplice e si puo' capire guardando `file.i`.

  I file `file.c` e `file.i` devono avere lo stesso nome base.

  L'estensione `.i` specifica all'eseguibile `swig` che il file non deve essere precompilato da CPP.
- Eseguiamo il comando terminale:

  ```
  swig -python file.i
  ```

Vengono generati due file: `file_wrap.c` serve per creare la libreria C, `my_library.py` e' ridondante e serve da interfaccia: se non si vuole includere direttamente la libreria C, che si chiamera' `_my_library.so`, si puo' importare `my_library.py`, che a sua volta importa `_my_library.so`, ma con maggiori controlli.

- Eseguiamo il comando terminale:

  `gcc -c file.c`

  Il comando consiste nella compilazione di `file.c` e nella creazione dell'eseguibile `file.o`.

  L'opzione `-c` rimuove l'operazione di LINKING.

- Eseguiamo il comando terminale:

  `gcc -fPIC -c file_wrap.c -I/usr/include/python2.7`

  Il comando consiste nella compilazione di `file_wrap.c` e nella creazione dell'eseguibile `file_wrap.o`.

  L'opzione `-fPIC` e' necessaria ma non si sa a cosa serve.

  L'opzione `-c` rimuove l'operazione di LINKING.

  L'opzione `-I/usr/include/python2.7` include nel path del preprocessore CPP la cartella che contiene il file `Parser.h`. Questa cartella potrebbe avere dei percorsi diversi a seconda della macchina.

- Eseguiamo il comando terminale:

  `ld -shared file.o file_wrap.o -o _my_library.so`

  Il comando consiste nell'unione degli eseguibili `file.o`, `file_wrap.o` e nella creazione della libreria `_my_library.so`, che li comprende entrambi.

  L'opzione `-shared` serve per mettere assieme i due eseguibili.

  L'opzione `-o` serve per specificare il nome della libreria di output.

  E' necessario che il nome base della libreria sia uguale a quella del file `my_library.py` con un UNDERSCORE _ come prefisso.

  Dopo la creazione della libreria `_my_library.so` gli eseguibili `file.o`, `file_wrap.o` possono essere cancellati, non servono piu'.

- Prendiamo il file PYTHON dove vogliamo importare la libreria C, nel nostro caso `test.py`.

  All'interno di questo file e' possibile importare equivalentemente la libreria C `_my_library.so` o l'interfaccia PYTHON `my_library.py`.

  Solitamente e' preferibile importare l'interfaccia PYTHON `my_library.py`.

12 Parser

Parser. Un PARSER e' un programma che prende in input una stringa – per esempio il contenuto di un file – e restiuisce il suo albero sintattico.
L'albero sintattico e' formato da nodi, e i nodi contengono delle informazioni.

Attenzione. Esistono piu' metodi per creare un parser. Qui presentiamo uno dei metodi piu' diffusi.

Definizione. La definizione di un PARSER si divide in due parti: la definizione del lessico e la definizione della sintassi.

Presentazione. Esistono due modi per descrivere la definizione di un parser, uno astratto e uno tecnico.

12.1 Descrizione astratta

Definizione. Il lessico viene definito usando un programma chiamato LEXER, la sintassi viene definita usando un programma chiamato GENERATORE DI PARSER.

Un parser non viene definito esplicitamente, ma viene generato automaticamente in base alle regole specificate al LEXER e al GENERATORE DI PARSER.

Lexer. Il LEXER e' un programma che prende in input una stringa – per esempio il contenuto di un file – e la divide in una lista stringhe primitive, in base alle regole grammaticali che gli vengono specificate.

Generatore di parser. Il GENERATORE DI PARSER e' un programma che prende in input una lista di stringhe e costruisce il corrispondente albero sintattico, in base alle regole sintattiche che gli vengono specificate.

12.2 Descrizione tecnica

Definizione. Il lessico viene definito usando un programma chiamato LEXER, la sintassi viene definita usando un programma chiamato GENERATORE DI PARSER. Questi due programmi sono stati creati apposta per essere usati assieme: la loro funzione combinata e' quella di creare un parser e non ha senso usarli separatamente.

Un parser non viene definito esplicitamente, ma viene generato automaticamente in base alle regole specificate al LEXER e al GENERATORE DI PARSER.

Programmi. Il LEXER che usiamo si chiama FLEX. Il GENERATORE DI PARSER che usiamo si chiama BISON.

12.2.1 Flex

Flex. FLEX e' un programma che prende in input un file `input.l`, su cui e' definito il lessico, e restituisce un file `input.c` che implementa una funzione chiamata PROSSIMO TOKEN.

Prossimo token. La funzione PROSSIMO TOKEN ad ogni chiamata restituisce il TOKEN associato alla stringa primitiva successiva.

Token. Un TOKEN e' una struttura che contiene le informazioni riguardanti una stringa primitiva.

input.l. All'interno del file `input.l` viene specificata il lessico, cioe' quali sono le stringhe primitive e quali sono le informazioni che vanno aggiunte ai rispettivi TOKEN – genericamente ad ogni stringa primitiva viene associato un TOKEN.

12.2.2 Bison

Bison. BISON e' un programma che prende in input un file `grammar.y`, su cui e' definita la sintassi, e la funzione PROSSIMO TOKEN, che implementa il lessico, e restituisce una coppia di file `grammar.c`, `grammar.h`, che implementano il PARSER.

I file `grammar.c`, `grammar.h` contengono una funzione che implementa il PARSER. Il file `grammar.h` deve essere incluso nel file C dell'utente, e per eseguire il parsing all'interno del file dell'utente basta chiamare la funzione che parsa.

Albero sintattico. L'albero sintattico e' formato da NODI. Un NODO e' una struttura che contiene delle informazioni – genericamente ad ogni TOKEN viene associato un NODO.

grammar.y. All'interno del file `grammar.y` viene specificata la sintassi, cioe' le regole mediante cui dalla successione di TOKEN viene generato l'albero sintattico.

Nel file `grammar.y` vengono specificate le informazioni che vanno aggiunte ai NODI.

12.2.3 CMake

Compilazione. I passi della creazione del PARSER sono i seguenti:

(1) il programma FLEX viene invocato e la sua esecuzione prende in input il file `input.l` e genera i file `input.c`, `input.h`

(2) il programma BISON viene invocato e la sua esecuzione prende in input il file `grammar.y` e genera il file `grammar.c`

(3) l'unione dei file sorgenti C dell'utente e dei file `input.c`, `input.h`, `grammar.c` costituisce il progetto C completo e puo' essere compilato

Per non dover chiamare l'esecuzione di 3 programmi diversi – cioe' FLEX, BISON e il compilatore C – con CMAKE si puo' definire un unico script che esegue tutti e 3 i programmi.

In questo modo i file sorgenti del progetto sono esattamente:

- i file sorgenti C dell'utente
- il file `input.l`
- il file `grammar.y`

I file `input.c`, `input.h`, `grammar.c` non sono file sorgenti ma vengono generati ad ogni compilazione, e si trovano all'interno della cartella `build`.

Il vantaggio e' che i file `input.l`, `grammar.y` sono molto piu' compatti, comprensibili e manipolabili rispetto ai rispettivi file `input.c`, `input.h`, `grammar.c`, e questo consente di modificare le regole del PARSER con minore difficolta'.

12.2.4 Allegato

Nella cartella `parser_example` e' definito il PARSER di una calcolatrice.

13 Kratos

13.1 Concorrenza

Concorrenza: uno SCHEDULER implementa una concorrenza tra i TASK. Il modo in cui viene usato lo SCHEDULER consente o vieta modellizzazioni di tipo diverso.

Concorrenza con preempt: uno SCHEDULER manda il segnale `do` al TASK, ma durante l'esecuzione del TASK lo SCHEDULER puo' inviare il segnale `stop` (chiamato anche PREEMPT, o INTERRUPT), eseguire altre istruzioni, e poi mandare il segnale `resume` al TASK per far riprendere l'esecuzione del TASK da dove era stata interrotta.

Concorrenza cooperativa: uno SCHEDULER manda il segnale `do` al TASK e attende il segnale `done` dal TASK. Fino a quando non riceve il segnale `done`, lo SCHEDULER non puo' eseguire altre istruzioni e non puo' interrompere il TASK.

13.2 Kratos

ESST: la tecnica ESST – per esteso EXPLICIT SCHEDULER AND SYMBOLIC THREADS – e' una tecnica per modellizzare un codice che presenta uno SCHEDULER e dei TASK. Il modello prevede la definizione individuale dello SCHEDULER e dei TASK, e definisce la loro interazione mediante CONCORRENZA COOPERATIVA.

SystemC: e' un sottolinguaggio di C++ definito per descrivere PROCESSI CONCORRENTI.

Kratos: e' un programma che traduce un codice SYSTEMC in un modello NUXMV servendosi della tecnica ESST, cioe' implementando una concorrenza cooperativa dello SCHEDULER e dei TASK.

14 Siti Web

$ `wget -r` ⟨`url`⟩
- scarica in contenuto della pagina web specificata

$ `wget -r -H` ⟨`url`⟩
- scarica in contenuto della pagina web specificata e di tutte le sottopagine e i link, ricorsivamente

15 Cluster

Una **coda** e' un insieme di macchine, dette **nodi**. Il **cluster** e' un insieme di **code**.

Le code del cluster sono:

- **es.q**: riservata all'unita' ES, ciascun nodo ha 96 giga di RAM;
- **es2.q**: riservata all'unita' ES, ciascun nodo ha 48 giga di RAM;
- **bld.q**: condivisa da tutto FBK.

Ci sono due modi per usare il cluster:

- **qlogin**: si accede al cluster usando una shell interattiva;
- **qsub**: si aggiungono in coda al cluster dei comandi.

15.1 Qlogin

⟨user_machine⟩ $ `ssh korein` → ⟨`password`⟩
- accede alla zona di ingresso del cluster –non siamo ancora in una coda

ESEMPIO. `cube$ ssh korein`

korein $ `screen`

■ crea una nuova sessione

ESEMPIO. `korein$ screen`

korein $ `screen -ls`
■ elenca le sessioni aperte

ESEMPIO. `korein$ screen -ls`

korein $ `screen -r ⟨session_name⟩`
■ riprende la sessione gia' esistente specificata

ESEMPIO. `korein$ screen -r 8179.pts-81.korein`

korein $ `qlogin -q ⟨queue_name⟩ -l mf=⟨giga_number⟩G`
■ accede alla coda specificata allocando la quantita' di memoria specificata

ESEMPIO. `korein$ qlogin -q es.q -l mf=40G`

ESEMPIO. `korein$ qlogin -q es2.q -l mf=85G`

korein $ `Ctrl + A + D`
■ esce dalla sessione corrente senza ucciderla

korein $ `exit`
■ esce dalla sessione corrente uccidendola

⟨user_machine⟩ $ `scp [⟨machine_1⟩]: ⟨machine_1_path⟩ [⟨machine_2⟩]: ⟨machine_2_path⟩`
■ copia il file specificato nel primo path nella posizione specificata nel secondo path

ESEMPIO. `cube$ scp ./shared_folder/file korein: /es0/sbicego/shared_folder/file`

ESEMPIO. `cube$ scp korein: /es0/sbicego/shared_folder/file cube: /tmp/file`

korein_emacs $ `Ctrl+X+C`
■ chiude il file attualmente aperto con EMACS

15.2 Qsub

⟨user_machine⟩ $ `ssh korein` → ⟨password⟩
■ accede alla zona di ingresso del cluster –non siamo ancora in una coda

ESEMPIO. `cube$ ssh korein`

korein $ `qsub -q ⟨queue_name⟩ -l mf=⟨giga_number⟩G -N ⟨nickname⟩ -S ⟨executable⟩`
■ aggiunge alla coda specificata il processo specificato

ESEMPIO. `korein$ qsub -q es.q -l mf=40G -S ./nuXmv -source cmds /es0/sbicego/model.smv`

ESEMPIO. `korein$ qsub -q es.q -l mf=40G -S /bin/bash /es0/sbicego/script.sh`

L'esecuzione di un processo produce due file:

- ⟨executable_or_nickname⟩.o⟨process_name⟩: lo standard output;
- ⟨executable_or_nickname⟩.e⟨process_name⟩: lo standard error.

ESEMPIO. `nuXmv.o868921`

korein $ `qstat -q ⟨queue_name⟩ -f`
- elenca le macchine e i processi della coda specificata

ESEMPIO. `korein$ qstat -q es.q -f`

korein $ `qdel ⟨process_name⟩`
- uccide il processo specificato

ESEMPIO. `korein$ qdel 868920`

16 Web

⟨web_page⟩ → `F5`
- ricarica la pagina senza ricaricare la cash

⟨web_page⟩ → `Ctrl+Shift+R`
- ricarica la pagina