UNIVERSITÀ DEGLI STUDI DI

MODENA E REGGIO EMILIA

Dipartimento di Ingegneria “Enzo Ferrari”

Corso di Laurea in Ingegneria Informatica

LA SHELL TESTUALE SMALLSH IN UNIX: ANALISI DELLA VERSIONE ESISTENTE E IMPLEMENTAZIONE DI ALCUNE ESTENSIONI

Tutor:

Chiar.mo Prof. Letizia Leonardi

Laureando:

Filippo Garagnani

Anno Accademico 2023-2024

Ringraziamenti

Desidero dedicare questo spazio a tutti coloro che hanno contribuito alla realizzazione del presente elaborato e che mi hanno sostenuto durante questa prima parte del mio percorso accademico.

In primo luogo, è doveroso esprimere un ringraziamento speciale al mio tutor, la Prof.ssa Letizia Leonardi, per la sua estrema disponibilità, la sua guida preziosa e la sua instancabile pazienza. Grazie alla sua competenza ed il suo supporto, questo lavoro rimarrà, per me, un'esperienza formativa ed estremamente arricchente.

Un altro importante ringraziamento va ai miei familiari, per il loro inesauribile sostegno morale e affettivo. In particolare ai miei genitori, per avermi fornito tutto il supporto necessario per affrontare questo percorso.

Vorrei anche ringraziare i miei amici e i miei colleghi di studio: la loro presenza è stata fondamentale per poter camminare con serenità e leggerezza. Hanno reso questo cammino molto meno arduo.

Infine, desidero ringraziare i docenti che ho avuto la fortuna di incontrare lungo tutto il mio percorso formativo. La loro dedizione e il loro impegno nel trasmettere conoscenze sono stati determinanti per la mia crescita, non solo accademica.

Indice

Introduzione **6**

1 Le funzionalità di smallsh **8**

1.1 Invocazione di smallsh 8

1.1.1 Parametri 8

1.1.2 Input e output 8

1.1.3 Attesa del comando 8

1.2 Analisi del comando 9

1.2.1 Argomenti 9

1.2.2 Metacaratteri 9

1.3 Esecuzione di un comando 10

1.3.1 Comandi built-in 10

1.3.2 Invocazione di eseguibili 11

1.4 Esecuzioni particolari 12

1.4.1 Esecuzione in background 12

1.4.2 Sequenze di comandi 12

1.4.3 Piping di comandi 12

2 Espansione dei percorsi **13**

2.1 I metacaratteri jolly 13

2.2 Implementazione in bash 13

2.3 Implementazione in smallsh 14

2.3.1 L'algoritmo di matching 14

2.3.2 L'espansione 16

2.3.3 Gestione casi d'errore 18

2.3.4 Confronto con bash 19

3 Variabili **20**

3.1 Implementazione in bash 20

3.2 Implementazione in smallsh 21

3.2.1 La struct variable 21

3.2.2 Creazione 21

3.2.3 Visualizzazione 23
3.2.4 Modifica 23
3.2.5 Eliminazione 24
3.2.6 Export 25
3.2.7 Variabili speciali 25
3.2.8 Gestione casi d'errore 26

4 Altre funzionalità **27**
4.1 Comando 'cd' 27
4.2 Inibizione 27
4.3 Comando 'eval' 29
4.4 Esecuzione di script 30
4.5 Espansione dei comandi 31
4.6 Stampa del prompt 32

5 Esempi d'uso **35**
5.1 Espansione dei percorsi 35
5.1.1 Il metacarattere '*' 35
5.1.2 Il metacarattere '?' 36
5.1.3 Elementi nascosti 37
5.2 Utilizzo di variabili 37
5.2.1 Gestione di variabili locali 37
5.2.2 Gestione di variabili d'ambiente 39
5.3 Comando 'cd' 40
5.4 Inibizione 41
5.5 Comando 'eval' 43
5.6 Esecuzione di script 44
5.7 Espansione dei comandi 45
5.8 Stampa del prompt 46

6 Conclusioni **47**

Appendici **48**
A Appendice A: La libreria 'dirent.h' 48

A.1 Le struct ‘DIR’ e ‘dirent’ 48
A.2 Apertura di directory 48
A.3 Lettura dello stream 49
A.4 Chiusura di directory 50
A.5 Tipologie di dirent 50
B Appendice B: Le funzioni per variabili d’ambiente 51
B.1 Recupero valore di variabili d’ambiente 51
B.2 Aggiunta e modifica di variabili d’ambiente 52
B.3 Rimozione di variabili d’ambiente 53
C Appendice C: Differenze implementative di bourne, bash e smallsh 54
C.1 Redirezione 54
C.2 Piping di comandi 54
C.3 Esecuzioni particolari 55
C.4 Espansione dei percorsi 56
C.5 Sostituzione delle variabili 56

7 Bibliografia **57**

Introduzione

Il presente elaborato è stato scritto per documentare il lavoro di estensione di un programma che realizza una shell molto semplice - chiamata *smallsh* - per i sistemi UNIX; la versione da cui si è partiti risaliva al 26 Novembre 2001.

In particolare, la versione di partenza prendeva spunto da un esercizio presente nel libro 'UNIX System Programming' [23] denominato anch'esso smallsh. La prima release è stata redatta dalla Prof.ssa Leonardi insieme con un collega dell'Università di Bologna. Dopo questa prima release, sono state prodotto altre realase da parte di alcuni collaboratori della Prof.ssa Leonardi, fino alla release da cui ha preso spunto il lavoro presentato in questo elaborato.

Il lavoro di estensione di *smallsh* si è focalizzato principalmente sull'implementazione di ulteriori funzionalità, senza alcuna modifica alla struttura del software originario. Gli arricchimenti apportati sono stati introdotti per rendere *smallsh* più vicina agli standard di altri programmi di shell moderni. La scelta di questo argomento è stata dettata dall'obiettivo di raggiungere una maggiore conoscenza sul funzionamento in generale di shell in UNIX.

A seguito della fase di analisi del codice sorgente, è stato svolto un processo di progettazione delle nuove funzionalità per un'esperienza utente migliorata; l'implementazione si è poi svolta in un'ottica di futura manutenibilità del codice. L'elaborato, quindi, è incentrato sull'analisi delle funzionalità aggiunte e sulla giustificazione delle scelte implementative, in relazione ad altri programmi di shell, come la Bourne Shell e Bash.

L'elaborato si compone di 5 capitoli e 3 appendici.

Nel primo capitolo si presenta una panoramica del software originale (che verrà chiamato *smallsh* originale nel seguito) e del suo comportamento. Nel secondo capitolo si analizza l'implementazione della prima estensione introdotta, relativa all'espansione dei percorsi mediante metacaratteri jolly. Nel terzo capitolo si descrive la seconda estensione, rappresentata dall'implementazione della gestione sia di variabili locali, che d'ambiente. Nel quarto capitolo sono presentate ulteriori funzionalità introdotte - come il comando 'eval' e l'espansione dei comandi. Infine, il quinto capitolo riporta l'esecuzione di alcuni esempi di utilizzo delle nuove funzionalità introdotte.

L’elaborato presenta nelle appendici A e B degli approfondimenti sulle funzioni di libreria utilizzate per lo sviluppo, principalmente ‘dirent.h’ (Appendice A) e ‘stdlib.h’ (Appendice B). Inoltre, nell’appendice C si trova una breve analisi che esamina le differenze a livello implementativo e comportamentale tra la *smallsh* estesa, *Bourne* e *Bash*.

1 Le funzionalità di smallsh

L'eseguibile *smallsh*, punto di partenza del presente elaborato, realizza un'interfaccia utente di tipo testuale. L'utente può fornire comandi al programma, il quale li interpreta ed esegue operazioni in base ad essi. Tale tipologia di interfaccia viene comunemente detta *shell*, da cui il nome di *smallsh*. Salvo casi particolari, una volta terminata l'esecuzione di un comando, *smallsh* attende un nuovo input da parte dell'utente. La versione esaminata in questo capitolo di *smallsh* - indicata nel seguito come versione originale - risale al 26 Novembre 2001.

1.1 Invocazione di smallsh

1.1.1 Parametri

L'eseguibile *smallsh* può essere eseguito senza parametri. Accetta però alcune opzioni per l'avvio:

- **-v**: prima dell'attesa di un comando da parte dell'utente, viene stampata a video una stringa che informa circa la versione di *smallsh* attualmente in esecuzione.
- **-h**: prima dell'attesa di un comando da parte dell'utente, vengono stampate alcune informazioni utili: nello specifico, viene fornita la versione di *smallsh* corrente, seguita da una stringa che elenca le diverse opzioni di invocazione per il programma.

1.1.2 Input e output

Di default, *smallsh* imposta come ingresso ed uscita il file **/dev/tty**, ossia il terminale del processo corrente. È ovviamente possibile modificare i file di input e di output mediante la ridirezione durante l'esecuzione di un comando (si veda il par. 1.2.2).

1.1.3 Attesa del comando

All'invocazione dell'eseguibile *smallsh* viene generato un processo che verrà indicato come "processo di smallsh". Tale processo di *smallsh* rimane in attesa di un comando fornito dall'utente. Per segnalare questa fase di attesa, viene stampata la stringa "Comando>" (che funge da *prompt*) dopo la quale l'utente fornirà i comandi che richiede siano eseguiti.

L'attesa del comando termina non appena viene inserito un invio o l'EOF (*end-of-file*) da tastiera.

1.2 Analisi del comando

I comandi che l'utente può fornire sono di natura testuale. Affinché la stringa possa essere correttamente eseguita, *smallsh* deve poter interpretarla, come svolto da qualunque shell. In generale, un comando è costituito da *argomenti* e *metacaratteri*.

1.2.1 Argomenti

Gli argomenti sono costituiti da sequenze di caratteri. Due argomenti adiacenti devono essere separati almeno da uno spazio o da una tabulazione; la presenza di più spazi o tabulazioni viene ignorata. Il primo argomento della stringa del comando deve rappresentare un comando che la shell sia in grado di gestire (si veda il paragrafo 1.3 per ulteriori dettagli). Eventuali altri argomenti che seguono il primo saranno interpretati come parametri per l'esecuzione del comando.

1.2.2 Metacaratteri

I metacaratteri rappresentano particolari casi di esecuzione del comando. Nella versione originale di *smallsh* sono i seguenti:

- '**;**': indica la possibile presenza sulla stessa riga di più comandi separati fra loro. Il comando prima del metacarattere viene eseguito per primo. Indipendentemente dall'esito dell'esecuzione - salvo l'eventuale terminazione del processo di *smallsh* -, viene eseguito il comando dopo il metacarattere. Più coppie di comandi separati da metacaratteri ';' possono essere concatenate fra loro.

- '**&**': posto alla fine di un comando ne richiede l'esecuzione in *background*. In sostanza, la presenza di tale metacarattere richiede che il processo di *smallsh* non attenda il termine dell'esecuzione del comando per riprendere a ricevere input dal parte dell'utente, ripresentando subito il *prompt* dei comandi.

- '|': richiede l'esecuzione di due comandi in *piping* fra loro. Viene creata una pipe che pone in comunicazione i due comandi: l'output del primo comando scrive

sulla pipe, mentre il secondo comando legge da essa. Ovviamente, più coppie di comandi separati da metacaratteri '|' possono essere concatenati fra loro, portando alla creazione di più pipe.

- '>': indica esplicitamente la ridirezione in output del comando che lo precede. L'argomento che segue il metacarattere deve rappresentare un file: su esso verrà scritto lo standard output del comando. Il file indicato viene sovrascritto. Se due metacaratteri '>' sono posti in sequenza, lo standard output del comando viene scritto in append al file.

- '<': indica esplicitamente la ridirezione in input del comando che lo precede. L'argomento che segue il metacarattere deve rappresentare un file: da esso verrà letto lo standard input del comando.

1.3 Esecuzione di un comando

1.3.1 Comandi built-in

Il primo argomento di un comando fornito da un utente deve indicare il comando che deve essere eseguito da *smallsh*. Nella maggior parte dei casi, si tratta di un'operazione che viene delegata ad un processo figlio che esegue un binario presente all'interno del sistema (si veda il par. 1.3.2). Vi sono, però, alcuni comandi particolari - detti *built-in*. Questi vengono gestiti ed eseguiti direttamente dal processo di *smallsh*. Nella versione originale, essi sono:

- **cd**: il comando built-in 'cd' - acronimo di *change directory* - si occupa di modificare la directory corrente. Richiede un singolo parametro che rappresenta il path relativo o assoluto a cui spostarsi. Nel caso in cui non venga passato il parametro o nel caso in cui il parametro non rappresenti un path valido, l'errore viene segnalato a video. (Per il comportamento del comando nella nuova versione, si veda il paragrafo 4.1).

- **lo**: il comando built-in 'lo' - il cui nome è il diminutivo di 'logout' - termina il processo di *smallsh*. Non richiede parametri aggiuntivi; nel caso in cui vengano forniti, sono ignorati.

- **ver**: il comando built-in 'ver' restituisce a schermo le informazioni della versione di *smallsh* in utilizzo. Non richiede parametri aggiuntivi; nel caso in cui vengano

forniti, sono ignorati. Le stesse informazioni sono fornite se si invoca *smallsh* con l'opzione '-v', come indicato nel paragrafo 1.1.1.

1.3.2 Invocazione di eseguibili

Dopo aver verificato che il primo argomento non corrisponda ad un comando built-in, si suppone l'esistenza di un eseguibile da invocare. Il nome di tale eseguibile deve coincidere con il primo argomento passato. Per eseguire questa tipologia di comandi, viene utilizzata la funzione **execvp** contenuta nella libreria unistd. La sua definizione [21] è:

```
#include <unistd.h>
int execvp(const char *file, char *const argv[]);
```

La funzione richiede come primo parametro ('file') una stringa che indichi il nome dell'eseguibile da invocare. Come secondo parametro richiede un array di stringhe: gli eventuali parametri per l'esecuzione del programma. Per convenzione, il primo di questi parametri deve corrispondere al nome del comando da cercare. Inoltre, la lista dei parametri deve concludersi con un puntatore a NULL.

La famiglia delle primitive **exec** presenta una particolarità: modifica la sezione di codice del processo attualmente in esecuzione. La sostituzione avviene, ovviamente, con il codice dell'eseguibile richiesto dall'utente. A causa di questo, eventuale codice rimanente del processo che invoca execvp non viene normalmente eseguito. Si ha però che *smallsh* - in qualità di shell - deve rimanere in esecuzione anche dopo l'esecuzione di comandi, a meno che l'utente non ne richieda la terminazione. Per garantire che *smallsh* non termini, come per tutte le shell, il processo di *smallsh* quindi genera un processo figlio: sarà questo processo ad invocare la funzione execvp. In questo modo, il processo padre - ossia, di *smallsh* - potrà proseguire nell'interpretazione ed esecuzione di comandi dopo aver atteso la terminazione del processo figlio, a meno che l'esecuzione non debba essere svolta in background (si veda il par. 1.4.1).

A meno che non venga fornito l'intero percorso del programma da invocare, la ricerca del binario da parte della funzione execvp avviene mediante la variabile d'ambiente 'PATH'.[1] Questa contiene una sequenza di directory, separate da ':', in cui viene cercato l'eseguibile.

[1] Si supponga, da qui innanzi, che la variabile 'PATH' contenga il carattere '.', ossia il riferimento alla directory corrente.

Se l'eseguibile non viene trovato, la funzione ritorna: il processo figlio allora stampa a video l'errore, terminando con -1.

1.4 Esecuzioni particolari

In presenza di alcuni modificatori, *smallsh* gestisce casi particolari di esecuzione.

1.4.1 Esecuzione in background

Se al termine del comando è presente il metacarattere **&**, si richiede l'esecuzione del comando in background. In questo caso il processo padre (*smallsh*) stampa a video il PID del processo figlio generato, ma non lo attende: il processo figlio esegue, appunto, in background.

1.4.2 Sequenze di comandi

La presenza del metacarattere **;** richiede l'esecuzione di due comandi in ordine. Durante il parsing della stringa fornita dell'utente, se viene trovato il metacarattere ';', si esegue quanto analizzato a quel punto come se fosse un comando a sé stante. Una volta terminata l'esecuzione, si considera la parte rimanente della stringa come se fosse un nuovo input da parte dell'utente.

1.4.3 Piping di comandi

In presenza del metacarattere |, si richiede l'utilizzo di una pipe tra due comandi. Se il metacarattere si trova all'inizio della stringa di comando, si ha un errore: il primo comando non può essere vuoto. Altrimenti, il processo figlio del processo di *smallsh* creerà la pipe. Verrà quindi creato un figlio, a cui viene affidata l'esecuzione del comando precedente al metacarattere, mentre il padre (cioè il figlio del processo di *smallsh*, a causa della natura del metodo execvp) eseguirà il comando successivo. Ovviamente, il figlio avrà come file di output non quello di default - /dev/tty - ma il lato di scrittura della pipe. In maniera analoga, il padre leggerà in input dal lato di lettura della pipe. Per garantire la possibilità di utilizzo di più comandi in piping, viene utilizzata la funzione che si occupa di eseguire il comando in maniera ricorsiva.

2 Espansione dei percorsi

A differenza di altre shell, *smallsh* non implementava, nella versione originale, l'**espansione dei metacaratteri jolly**, cioè caratteri particolari mediante i quali è possibile espandere una stringa a uno o più nomi di file e directory.

2.1 I metacaratteri jolly

I principali metacaratteri jolly utilizzati per l'espansione dei percorsi sono:

- *****: il metacarattere jolly '*' si espande ad una qualsiasi stringa, inclusa la stringa vuota. Ad esempio, la stringa 's*' si può espandere a qualsiasi nome che inizi con il carattere 's'.

- **?**: il metacarattere jolly '?' si espande ad uno ed un solo carattere qualsiasi. Ad esempio, la stringa 'main.?' si può espandere a 'main.c' o 'main.h', ma non a 'main.exe' o a 'main.'.

- **[]**: i metacaratteri jolly '[]' si espandono a qualsiasi sequenza composta da e solo da i caratteri contenuti tra le due parentesi quadre. L'utilizzo del simbolo '-' all'interno delle parentesi segnala la presenza di un *range* di caratteri. Ad esempio, '[a-z]' si espande a qualsiasi nome composto unicamente da lettere minuscoli; invece, '[a1]' si espande a qualsiasi nome composto unicamente dai caratteri 'a' e '1'.

2.2 Implementazione in bash

Dalla documentazione di *bash* [5]: "After word splitting [...] Bash scans each word for the characters '*', '?', and '['. If one of these characters appears, and is not quoted, then the word is regarded as a pattern, and replaced with an alphabetically sorted list of filenames matching the pattern. If no matching filenames are found [...] the word is left unchanged. [...] When matching a filename, the slash character must always be matched explicitly by a slash in the pattern [...]".

La documentazione rivela, dunque, come *bash* gestisce l'espansione dei percorsi. Solo dopo aver raccolto gli argomenti e i metacaratteri di un comando, controlla uno ad uno gli elementi passati. Se trova un carattere jolly, e l'elemento non è fra virgolette, allora sostituisce il pattern con una lista (ordinata alfabeticamente) di percorsi che rispettino il

pattern; se il pattern non si può espandere a nessun percorso, viene trattato come una stringa. Rivela anche che il carattere '/' in un percorso deve avere una corrispondenza con un carattere '/' nel pattern, e non con un carattere jolly.

2.3 Implementazione in smallsh

Per implementare l'espansione dei percorsi in *smallsh* si è deciso di includere e supportare i metacaratteri jolly '*' e '?', nonché la possibilità di inibire l'espansione mediante l'uso di virgolette, sia singole che doppie (si veda il par. 4.2).

In base a come già *smallsh* svolgeva le sue operazioni, è stato deciso come implementare i passaggi necessari per l'espansione dei percorsi. Ogni volta che un argomento viene individuato, dopo averlo salvato, si verifica se esso è espandibile. Si controlla dunque che non sia fra la stessa tipologia di virgolette - le quali sono sempre rimosse - e se presenta metacaratteri jolly. Se non è espandibile, *smallsh* opera regolarmente e procede nell'analisi del successivo elemento della stringa di comando. Altrimenti, si determina la directory da cui iniziare la ricerca: se la stringa del pattern inizia con '/', l'espansione avrà inizio dalla root del sistema; altrimenti, dalla directory corrente. In maniera ricorsiva si procede poi a cercare match e a salvare ognuno di essi come argomento. In breve, la stringa contenente il pattern viene sostituita da un numero di argomenti pari al numero di match trovati. Una volta terminata la ricerca, se non si è trovato alcun match, si considera la stringa passata come una stringa non da espandere. Fatto ciò, *smallsh* riprende nell'analisi degli altri eventuali argomenti successivi del comando.

2.3.1 L'algoritmo di matching

Per garantire l'estensibilità nel futuro di *smallsh*, si è deciso di fare uso di due funzioni: la prima per verificare se una stringa è espandibile e la seconda per validare che una stringa corrisponda ad un pattern. In questo modo, basterebbe modificare questi due metodi per l'utilizzo di ulteriori metacaratteri jolly.

Per controllare se un argomento può essere espanso, si invoca la funzione:

```
int contain_matching(char* string);
```

Il valore di ritorno è 1 se la stringa passata come parametro è espandibile, 0 altrimenti. La funzione opera nel seguente modo: si controlla la stringa passata, confrontando carattere

per carattere con gli elementi di un array statico. L'array contiene i metacaratteri jolly, terminati da '/0'. Se la stringa presenta almeno un carattere jolly, allora la stringa è espandibile e il valore di ritorno è 1. La complessità di questo metodo è $O(2n) = O(n)$, dove 2 è il numero di caratteri nell'array statico ed n il numero di caratteri nella stringa.

Per verificare che una stringa faccia match con un pattern fornito, sono state definite le funzioni:

```
int is_pattern_match(char *pattern, char *string){
    if(pattern[0] == '\0' && string[0] == '\0')
                return 1;
    if(pattern[0] == '*')
                return is_match_star(pattern + 1, string);
    if(string[0] != '\0' &&
        (pattern[0] == '?' || pattern[0] == string[0]) )
                return is_pattern_match(pattern + 1, string + 1);
    return 0;
}

int is_match_star(char* pattern, char* string){
        if(string[0] == '\0' || pattern[0] == '\0') return 1;
        while(string[0] != '\0'){
                if(is_pattern_match(pattern, string)) return 1;
                string += 1;
        }
        return 0;
}
```

La seconda funzione è ausiliare alla prima. In sostanza, è il valore di ritorno della prima che rivela se il parametro 'string' fornisce un match rispetto al parametro 'pattern' (1 se vero, 0 se falso). Si consideri, per iniziare, la seconda funzione. Questa suppone che il pattern sia del tipo '*[contenuto del parametro 'pattern']'. Se entrambe le stringhe sono vuote, si ha un match ('pattern' è '*', mentre 'string' è vuota; per le regole dei metacaratteri jolly ciò è un match). Altrimenti, si suppone che l'asterisco si espanda dalla stringa vuota, progressivamente, a tutti i caratteri di 'string'. Si invoca dunque

'is_pattern_match': se si ha un match, si ritorna 1. In caso contrario, si incrementa 'string' di 1 (si suppone che l'asterisco includa anche il carattere successivo) e si procede, fino a che 'string' non diventi eventualmente vuota. In quel caso, non avendo trovato match, si ritorna 0.

La prima funzione considera tre casi. Il primo caso valuta se entrambe le stringhe siano vuote (ossia, se puntano entrambe al carattere '/0'): se così è, allora si ha un match. Il secondo caso è quello in cui 'pattern' inizi con '*': allora si ritorna il valore di 'is_match_star', invocato con 'pattern' incrementato di 1 (poiché la funzione suppone già che il pattern totale sia il parametro passato preceduto da un asterisco) e 'string'. Il terzo caso prevede che 'string' non sia nulla: allora, se 'pattern' possiede un '?' oppure se il primo carattere di 'string' e 'pattern' coincidono, si reinvoca 'is_pattern_match' con entrambe le stringhe incrementate di 1. Se non si verifica nessuno di questi tre casi, allora è garantito che non vi sia un match: la funzione ritorna 0.

2.3.2 L'espansione

Una volta determinato che una stringa deve essere espansa, vengono passati una serie di parametri alla funzione principale di questa fase:

```
int search_and_save(int* narg, int* j, char* tokbuf, char** arg,
                    int depth, char* pattern, char* possible_path,
                    int path_index, DIR* d);
```

Quattro parametri sono variabili globali di un file, e in quanto tali sono passate per riferimento affinchè possano essere modificate: sono **narg** (il numero degli argomenti esaminati), **j** (un indice che ricorda la posizione dell'ultimo elemento salvato in tokbuf), **tokbuf** (l'array che contiene gli argomenti e i metacaratteri, esclusi se espansi * e ?, separati da '/0') e **arg** (un array di stringhe che contiene un riferimento all'inizio di ogni elemento in tokbuf, eventualmente NULL se non si desidera modificarlo). Il parametro **depth** tiene conto del livello di profondità della ricerca a cui si è arrivati. Ovviamente, alla prima invocazione viene impostato a zero; ad ogni invocazione ricorsiva, aumenta di 1. La stringa **pattern** contiene la stringa fornita dall'utente di cui bisogna trovare tutti i possibili match. La stringa statica **possible_path** viene riempita mano a mano che si trovano 'candidati' per il pattern matching; una volta giunti alla profondità richiesta dalla stringa 'pattern' sarà riempita (se si trova un match) e copiata in tokbuf. La variabile **path_index**

tiene traccia del punto a cui si è giunti a copiare la stringa 'possible_path' - così come l'intero 'j' per tokbuf. Infine, la variabile **d** di tipo *DIR* (si rimanda all'Appendice A per un'analisi dettagliata della struct DIR e del suo utilizzo) contiene la directory da esplorare. Alla prima invocazione della funzione, 'd' corrisponderà alla root del sistema - se il pattern rappresenta un percorso assoluto - altrimenti alla directory di lavoro corrente. Il valore di ritorno è pari al numero di match totali trovati.

La funzione si basa sul fatto che ogni 'pattern' estendibile presenta una profondità, relativa alla directory di partenza. Ad esempio, "/bin*" e "test.?" sono pattern che presentano una profondità pari a 0: si trovano nella directory di inizio ricerca - rispettivamente la root e la directory corrente. Mentre "/bin/c*/*" è un percorso con profondità pari a 2. Analogamente, si può considerare parte del path estendibile basandosi sulla profondità. In questo modo, "/bin/c*/*" avrà parte di path con profondità relativa 1 la stringa "c*". Due funzioni distinte si occupano di determinare la profondità relativa e di ottenere la parte di path ad una determinata profondità:

```
char* get_relative_name(char* entire_path, int depth);
int nested_level(char* path);
```

La prima funzione ritorna un puntatore all'inizio della stringa con la parte del path intero ('entire_path') alla profondità richiesta ('depth'). La seconda funzione ritorna la profondità relativa della stringa 'path' passata.

La funzione 'search_and_save' controlla che la profondità di ricerca (la variabile 'depth') coincida con la profondità relativa della stringa 'pattern': si ha allora il caso base della ricorsione. Iterando su ogni elemento della directory corrente (passata attraverso il parametro 'd'), si controllano eventuali match tra la parte finale del pattern e il nome. In particolare, se il pattern termina con uno '/' si verificano solamente i nomi di directory (si noti che questa specifica è presente solo in *bash*, e non in *bourne*). Se si verifica un match, si completa l'array statico 'entire_path' e se ne salva l'intero contenuto in 'tokbuf', incrementando opportunamente anche 'narg' e 'j'. Prima di aggiungere gli elementi agli array, si controlla ovviamente che essi contengano spazio a sufficienza. Ritorna poi il valore della variabile locale 'matched', la quale contiene il numero totale di match ottenuti.

Se viene invocata la funzione ma 'depth' è minore della profondità relativa del pattern, allora è necessario svolgere delle operazioni per proseguire nella ricerca ad un livello di profondità maggiore. Nella directory passata come parametro 'd' si cercano eventuali

directory il cui nome fa match con la parte relativa del pattern fornito. Se, ad esempio, 'depth' è pari ad 1 e 'pattern' contiene la stringa "/bin/c*/*", in questa fase si cercheranno tutte le directory il cui nome fa match il pattern "c*". Quando si trova un match, si integra nel parametro che tiene traccia del percorso che si sta delineando e si invoca nuovamente la funzione 'search_and_save', aumentando 'depth' di 1, usando ricorsivamente la directory trovata. Si incrementa poi una variabile locale 'matched' del valore di ritorno della chiamata ricorsiva. Si elimina dal parametro con il percorso totale il nome dell'ultima directory e si ripete il procedimento sugli elementi rimasti all'interno della directory.

Indipendentemente dai casi, la funzione svolge alcune operazioni sempre uguali: determina, prima di iniziare, la parte del pattern da controllare e ritorna, in ogni caso, il valore della variabile locale 'matched'.

Un caso particolare è dato dagli elementi nascosti (incluse le directory '.' e '..'). Infatti, *bash* mostra tali elementi solo se è presente il carattere '.' all'inizio del pattern. Si è deciso di adottare la stessa convenzione per *smallsh*. Ad ogni invocazione di 'search_and_save' si controlla se la parte del path relativa alla profondità raggiunta presenti o meno '.' come primo carattere. Se così non fosse, ogni elemento nascosto viene ignorato. Altrimenti, tutti gli elementi nascosti vengono presi in considerazione. Ovviamente, l'espansione o meno degli elementi nascosti dipende dal match con il pattern fornito.

L'espansione non avviene nel caso in cui i metacaratteri jolly siano inibiti attraverso l'uso di apici singoli o doppi (si veda il paragrafo 4.2).

2.3.3 Gestione casi d'errore

Durante l'espansione, possono verificarsi diverse tipologie di errore. La modalità di gestione di questi è stata decisa in base alla gravità del problema e alla sua natura.

- **Lunghezza del percorso**: si è supposta la lunghezza massima di un percorso pari a 512 caratteri - ossia la metà dei caratteri massimi consentiti per un comando nella versione originale di *smallsh*. Questo valore è la dimensione dell'array statico usato dalla funzione dell'espansione dei percorsi per conservare progressivamente i path possibili per cui si ha un match. Se si tenta di salvare più caratteri di quelli richiesti dalla dimensione, viene stampato a video un errore e non viene aggiunto il parametro all'array.

- **Lunghezza del comando espanso**: il comando fornito dall'utente, contenuto nella

variabile globale ‘tokbuf’, non deve superare i 1024 caratteri. Anche in questo caso, se a causa dell’espansione dei percorsi, gli argomenti superano questo valore, un errore viene stampato a video e l’argomento non viene salvato.

- **Numero degli argomenti**: l’array ‘arg’ - che contiene i riferimenti agli inizi dei diversi comandi e metacaratteri contenuti in ‘tokbuf’ - possiede anch’esso un valore massimo, stabilito nella versione originale di *smallsh*, pari a 112 puntatori che può contenere. Il valore di ‘narg’ tiene dunque traccia del valore attuale di argomenti usati; se narg diventa maggiore o uguale a 112, allora - stampando a video un avvertimento a riguardo - non si procede con l’espansione del percorso. Questa scelta è motivata dal fatto che *smallsh* in origine imponeva questo limite al numero degli argomenti, gestito ignorando eventuali argomenti d’eccesso.

- **Apertura di directory**: in alcuni casi, la funzione ‘opendir’ nella libreria ‘dirent.h’ può fallire per diverse ragioni. Ad esempio, se non si dispongono delle autorizzazioni necessarie per l’apertura della directory oppure se è elevato il numero di file aperti dal processo. In ogni caso, se la funzione - invocata prima di richiamare ricorsivamente la funzione ‘search_and_save’ - fallisse, a video viene stampata una stringa che segnala l’errore. Si prosegue poi nella ricerca di altri elementi.

In generale si è preferito mantenere un approccio nella gestione degli errori che garantisse la possibilità di continuare nell’interpretazione di ulteriori comandi e nella non terminazione del processo di *smallsh*.

2.3.4 Confronto con bash

Come si è visto dalla documentazione di *bash*, ci sono alcune le differenze nell’implementazione dell’espansione dei percorsi tra le due shell. In primo luogo, l’ordine con cui i diversi percorsi vengono aggiunti al comando. In *bash*, l’ordine è alfabetico; in *smallsh*, l’ordine dipende dalla gestione dei file nel sistema UNIX [26]. In secondo luogo, *bash* implementa anche i metacaratteri jolly ‘[]’, a differenza di *smallsh*. Una terza differenza emerge nel momento in cui vengono individuati ed espansi i pattern: *bash* svolge questa operazione al termine dell’analisi del comando, laddove *smallsh* se ne occupa durante l’analisi della stringa, argomento per argomento. Infine, diversamente da come opera *bash*, *smallsh* impone una lunghezza massima ammessa del percorso.

3 Variabili

Nella maggior parte delle shell, possono essere definite dall'utente delle *variabili locali*. Queste sono identificate da un nome e possiedono un valore; questi elementi sono entrambi stringhe. Le shell che supportano l'uso di variabili locali, presentano le funzionalità di creazione e di eliminazione di variabili, nonchè - fornendo il nome della variabile - di assegnamento e di recupero del valore.

Le variabili locali, come suggerisce il nome, sono unicamente accessibili all'interno del processo della shell in uso. I processi figli della shell non possono in generale utilizzarle, a meno che esse non siano trasformate in *variabili d'ambiente*. I processi figli, infatti, ereditano per copia l'ambiente di processo del processo padre che li genera. All'interno di tale ambiente sono contenute anche le variabili d'ambiente. In generale, le shell implementano un comando *built-in* - chiamato solitamente **export** - mediante cui 'promuovere' una variabile locale, rendendola d'ambiente.

3.1 Implementazione in bash

Dalla documentazione *bash* [7]: "A variable is a parameter denoted by a name. A variable has a value and zero or more attributes. [...] A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the *unset* builtin command. A variable may be assigned to by a statement of the form 'name=[value]'. If value is not given, the variable is assigned the null string. [...] the '+=' operator can be used to append to or add to the variable's previous value."

La documentazione rivela come *bash* supporta l'uso di variabili locali. Una variabile è un parametro (un valore), identificato da un nome. Eventualmente, in *bash*, una variabile potrebbe possedere anche degli attributi. Per creare una variabile ed assegnarle un valore, l'espressione deve essere della forma 'nome=[valore]': in questo caso, 'valore' è facoltativo; se non viene fornito un valore, si assegna alla variabile la stringa vuota. Una variabile può essere rimossa solo con il comando *unset*. Infine, con l'operatore '+=' si può aggiungere in append una stringa al valore già posseduto della variabile.

3.2 Implementazione in smallsh

Molte funzionalità implementate in *bash* sono state adattate ed integrate all'interno di *smallsh*. Nei seguenti paragrafi, verrà spiegata la rappresentazione interna delle variabili locali e la modalità di gestione delle diverse funzionalità riguardanti le variabili, sia locali che d'ambiente. Per quest'ultime, sono state utilizzate alcune funzioni di 'stdlib.h'; si rimanda all'Appendice B per una loro descrizione più dettagliata.

3.2.1 La struct variable

Per quanto riguarda le variabili locali, è stata definita una *struct* apposita:

```
typedef struct variable {
    char* name;
    char* value;
    char reserved;
} variable;
```

I campi 'name' e 'value' contengono, rispettivamente, le stringhe del nome identificativo e del valore di una variabile locale. Il campo 'reserved' viene invece posto ad 1 per le variabili speciali non modificabili (si veda il par. 3.2.7), a 0 altrimenti. Per tenere traccia di tutte le variabili locali che l'utente può definire, *smallsh* conserva un array di puntatori al tipo 'struct variable'; inizialmente, ognuno dei puntatori dell'array è un riferimento a NULL. Si è deciso di porre un limite al numero di variabili locali pari a 100. Attraverso *smallsh*, l'utente è in grado di fornire comandi per inizializzare, modificare, visualizzare ed eliminare le variabili e i rispettivi valori.

3.2.2 Creazione

Per la creazione di variabili, si è adottata la sintassi di *bash*: l'utente deve fornire come comando built-in una stringa che deve rispettare il formato "nome=[valore]". Il campo 'nome' non può essere occupato dalla stringa vuota, a differenza del campo 'valore'. Per verificare che un comando rispetti il formato richiesto, si controlla - in primo luogo - che si abbia appunto un solo argomento nell'input fornito dall'utente. Si determina, poi, che tale argomento presenti una sola occorrenza del carattere '=' e che tale occorrenza non si trovi all'inizio della stringa. Per sapere, infine, se si tratta effettivamente di un'inizializzazione

si fa uso della funzione:

```
int var_exists(variable** variables, char* name){
        if(strcmp("$", name) == 0) return 1;
        for(int k = 0; k < MAXVAR; k++){
                if(variables[k] == NULL) continue;
                if(strcmp(variables[k]->name, name) == 0) return 1;
        }
        if (getenv(name) != NULL) return 1;
        return 0;
}
```

Come si nota, il valore di ritorno è '1' se la variabile esiste, altrimenti è '0'. In quest'ultimo caso, la variabile deve essere ovviamente creata. Per verificare che una variabile esista, si controlla ogni elemento non nullo dell'array di variabili locali: se esso presenta nel campo 'name' proprio il nome cercato, allora la variabile esiste ed è locale. Altrimenti, si fa riferimento al metodo 'getenv()' - una primitiva contenuta in 'stdlib.h'. La funzione controlla le variabili d'ambiente: se esiste una variabile con nome coincidente alla stringa passata, ritorna un puntatore al suo valore. Altrimenti, ritorna un puntatore a NULL. Le ultime due righe verificano appunto che la variabile richiesta non esista all'interno dell'ambiente.

Per creare una variabile, viene invocata la funzione:

```
void set_variable(variable** variables, char* name, char* value);
```

La funzione verifica, in primo luogo, che l'array globale di puntatori a *variable* non abbia raggiunto la sua capienza massima 'MAXVAR' (pari a 100, come descritto nel paragrafo 3.2.1). Si crea poi una struct di tipo variable, viene allocata la memoria per il suo nome 'name' e per il suo valore 'value' - entrambi copiati e non indirizzati direttamente. Il campo 'reserved' viene invece posto a 0. Infine, al primo puntatore a NULL nell'array globale viene associato il puntatore alla variabile appena creata.

3.2.3 Visualizzazione

Per visualizzare a video una variabile dato il suo nome, l'utente può utilizzare il metacarattere '$'. Esso può trovarsi in una qualsiasi posizione di un argomento: ciò che lo precede rimane inalterato, mentre ciò che segue viene inteso come il nome di una variabile e sostituito con il suo valore. Se ciò che segue il metacarattere non rappresenta il nome di una variabile - locale o d'ambiente - esistente, il tutto viene sostituito dalla stringa vuota.

L'implementazione della visualizzazione del valore di una variabile ricalca quanto fatto per l'espansione dei percorsi. Una volta salvato un argomento durante l'analisi del comando, se viene trovato almeno un metacarattere '$' - seguito da almeno un altro carattere diverso dal terminatore di stringa '\0' - si calcola il numero di invocazioni presenti[2] e per ognuna di esse viene salvato il nome della variabile invocata. Se in 'tokbuf' - che conserva gli argomenti e i metacaratteri del comando - è presente spazio sufficiente, si cancellano le invocazioni dalla stringa (partendo dal primo metacarattere '$') e si aggiungono, in ordine, i valori delle variabili richieste. Il metacarattere '$' può però essere inibito mediante l'uso di apici singoli: terminata l'analisi del comando, il metacarattere e ciò che lo segue non sono sostituiti e gli apici singoli sono rimossi se racchiudono l'argomento per intero.

Per ottenere il valore di una variabile, dato il nome, si utilizza la funzione:

```
char* get_var_value(variable** variables, char* name);
```

La funzione ritorna la stringa contenente il valore della variabile identificata dal nome 'name'. Viene subito verificato se la variabile è locale; nel caso, viene ritornato il campo 'value' della 'variable' interessata. Se, infine, la variabile è d'ambiente, si utilizza semplicemente il metodo 'getenv()' che, se non si verificano errori, ritorna proprio la stringa contenente il valore della variabile richiesta.

3.2.4 Modifica

La modifica del valore di una variabile - locale o non - può avvenire o in maniera diretta o in append. Nel primo caso, il valore della variabile viene completamente sovrascritto. La sintassi per questo tipo di modifica coincide con la sintassi della creazione di variabile:

[2] Si noti che il numero di invocazioni non coincide necessariamente con il numero di caratteri '$': infatti, se si vuole invocare la variabile locale con nome '$', servono due caratteri per una singola invocazione.

‘name=[value]’, dove ovviamente ‘name’ è il nome di una variabile esistente. Nel secondo caso, ossia di modifica in append, si aggiunge una stringa in coda al valore già posseduto dalla variabile. La sua sintassi è ‘name+=[value]’.

Quando un comando è costituito da una sola stringa - costituito da una delle due sintassi - e la funzione ‘var_exists()’ rivela che la variabile esiste, vengono invocate le funzioni relative alla modifica da applicare:

```
void edit_variable(variable** variables, char* name, char* str);
void append_variable(variable** variables, char* name, char* to_app);
```

Nella prima funzione, si controlla innanzitutto che la variabile sia locale e non speciale: in questo caso, ottenuto il riferimento mediante il nome ‘name’, si dealloca lo spazio usato dal precedente valore della variabile e se ne alloca nuovamente quanto necessario per la copia del nuovo valore, contenuto in ‘str’. Se la variabile è locale e speciale, allora la funzione ritorna senza modificare nulla. Se, al contrario, la variabile è d’ambiente, si delega tutto alla primitiva ‘setenv()’: fornito il nome della variabile d’ambiente, il nuovo valore che deve assumere e un booleano per la sovrascrittura del valore precedente, la funzione gestisce la modifica della variabile. Si è deciso di sovrascrivere sempre i valori delle variabili d’ambiente.

Anche nella seconda funzione si controlla subito se la variabile sia locale e non speciale: viene allora reallocato il campo di ‘value’, affinchè possa avere spazio sufficiente non solo per il valore precedente, ma anche per la stringa ulteriore da aggiungere. La stringa ‘to_app’ viene poi copiata in coda al campo contenente il valore della variabile. Come prima, variabili locali e speciali non vengono modificate in append. Se invece la variabile è d’ambiente, si ottiene - mediante ‘getenv()’ - il valore già posseduto dalla variabile. Si alloca spazio sufficiente per conservare entrambi i valori, copiati nella nuova zona di memoria, e la nuova stringa viene fornito come parametro al metodo ‘setenv()’, che la salva nell’ambiente. Anche in questo caso, si è stabilito di sovrascrivere i valori delle variabili d’ambiente.

3.2.5 Eliminazione

Il comando built-in *unset* permette di eliminare una variabile, che sia locale o d’ambiente. Il comando richiede un solo parametro aggiuntivo: il nome della variabile da eliminare. Se vengono forniti altri parametri, questi vengono ignorati. Se il nome fornito non corrisponde

ad una variabile, o corrisponde ad una variabile speciale, il comando non modifica nulla. Altrimenti si usa la funzione:

```
void unset_variable(variable** variables, char* name);
```

La funzione, in primo luogo, controlla se la variabile è locale. Nel caso, dealloca la memoria precedentemente usata dal puntatore a 'variable' - per i campi del nome e del valore. Ovviamente, la funzione rimuove la variabile dall'array globale di puntatori a 'variable'. Se la variabile è d'ambiente, la rimozione viene delegata alla primitiva 'unsetenv()': questa, fornito il nome della variabile, la elimina dall'ambiente.

3.2.6 Export

Il comando built-in *export* permette di 'promuovere' una variabile locale, rendendola d'ambiente. Il comando richiede un solo parametro aggiuntivo, ossia il nome della variabile da rendere d'ambiente. Se vengono forniti parametri ulteriori, questi vengono ignorati.

Il comando invoca la funzione:

```
void export_var(variable** variables, char* name);
```

Se non esiste alcuna variabile locale identificata dal 'name' fornito, o se la variabile è speciale, il comando non modifica nulla. Altrimenti, tenta di aggiungere la variabile all'ambiente, mediante la primitiva 'setenv()'. Se il suo valore di ritorno è 0, allora l'inserimento nell'ambiente è avvenuto con successo, e la funzione procede ad eliminare la variabile - attraverso la funzione 'unset_variable()' - dall'array di variabili locali.

3.2.7 Variabili speciali

Nelle operazioni eseguite da *smallsh* prima di attendere il primo comando da parte dell'utente, vengono allocate le risorse per la gestione delle variabili locali e vengono impostate le variabili speciali.

Le variabili speciali che vengono inizializzate sono:

- '$': contiene il PID del processo di *smallsh* in esecuzione.
- '?': contiene la stringa numerica dello stato di uscita dell'ultimo comando eseguito.
- '#': contiene il numero degli argomenti (oltre al nome dell'eseguibile) usati per invocare il processo di *smallsh*.

- '*': contiene una lista separata da spazi degli argomenti (escluso il nome dell'eseguibile) usati per invocare il processo di *smallsh*.

In *bash*, l'invocazione del processo di shell con argomenti non concessi risulta in un errore e nella terminazione immediata del processo [1]. Analogo è il comportamento di *smallsh* (si veda il par. 4.4). Di conseguenza, durante un'ordinaria esecuzione di *smallsh*, il valore delle due variabili '#' e '*' è rispettivamente 0 e "", così come per *bash*.

3.2.8 Gestione casi d'errore

È possibile riscontrare diverse tipologie di errore nella gestione delle variabili, locali o d'ambiente. Tra i casi più frequenti - con i rispettivi approcci - si notano:

- **Lunghezza del comando sostituito**: sostituendo ai nomi di variabile - preceduti da '$' - il valore della variabile, vi è la possibilità di ottenere una stringa di comando che superi la lunghezza massima consentita. Nel caso, viene stampato a video un messaggio che segnala l'errore e il comando non viene eseguito.

- **Utilizzo di variabili inesistenti**: se attraverso comandi built-in o attraverso il simbolo '$' l'utente cerca di riferirsi a una variabile inesistente - sia localmente che nell'ambiente - allora l'approccio scelto si basa sulla gestione operata da *bash*. In primo luogo, non vengono segnalati a video errori in nessun caso. Se l'utente tenta di visualizzare una variabile inesistente, si utilizza - come valore di tale variabile - la stringa vuota. Se l'utente usa un comando built-in, quali 'export' o 'unset', fornendo il nome di una variabile inesistente, allora il comando non svolge alcuna operazione.

- **Modifica dell'ambiente**: in alcuni casi particolari, l'uso delle primitive per la gestione dell'ambiente - quali 'setenv()' e 'unsetenv()' - possono fallire. Nel caso, una stringa mostra a video l'errore e l'operazione termina. Ovviamente, *smallsh* riprende la sua esecuzione attendendo un nuovo input dall'utente.

Come si nota, le scelte operate per la gestione degli errori si basano - laddove possibile - su come opera *bash*. Gli altri casi presentano comunque un approccio volto alla ripresa dell'esecuzione di *smallsh*, evitando una sua prematura terminazione.

4 Altre funzionalità

Oltre all'implementazione dell'espansione dei metacaratteri jolly e della gestione delle variabili - come si è visto nei precedenti capitoli - sono state aggiunte e integrate ulteriori funzionalità nella nuova versione di *smallsh*.

4.1 Comando 'cd'

Il comando built-in 'cd', presente nella versione originale di *smallsh*, è stato modificato per aumentarne le funzionalità. La prima modifica riguarda la variabile d'ambiente 'PWD' (acronimo di 'Printing Working Directory'): questa viene modificata ogni volta che il comando va a buon fine e la directory corrente viene cambiata. La seconda modifica riguarda, invece, l'invocazione del comando 'cd' senza argomenti: nella versione originale di *smallsh* questo portava a stampare un errore a video. Nella nuova versione, invece, si è deciso di adottare il comportamento che si ha in *bash* [2]: se non vengono forniti argomenti al comando 'cd', si assume che lo spostamento deve essere fatto verso la directory HOME, contenuta nell'omonima variabile d'ambiente. Nel caso in cui la variabile 'HOME' non dovesse esistere, viene stampato un errore a video e il comando non esegue nulla.

4.2 Inibizione

Nella versione originale di *smallsh*, non era presente la possibilità di inibire i metacaratteri forniti come comando. L'inibizione è stata aggiunta nella nuova versione mediante gli apici doppi e gli apici singoli.

Durante il parsing del comando fornito in input dall'utente, se viene individuata un'occorrenza di apice - sia singolo che doppio - tutto ciò che segue viene considerato un unico argomento fino alla successiva occorrenza della stessa tipologia di apice. Eventuali metacaratteri contenuti tra apici dello stesso tipo vengono inibiti e trattati come semplici caratteri. Se al termine dell'input fornito dall'utente non è stata trovata una seconda occorrenza di apici dello stesso tipo, si suppone la presenza di tale carattere al termine del comando. Visto

che la chiusura di apici, in un argomento, è necessaria[3], rispetto ad altre soluzioni questa è stata favorita per semplicità. Ad esempio, gli input:

```
Comando> var="var value"
```

```
Comando> var="var value
```

sono ritenuti identici da *smallsh*: nel secondo caso suppone che le doppie virgolette finali assenti siano implicitamente poste al termine del comando.

Entrambe le tipologie di apici permettono di inibire sia i metacaratteri 'originali', come ad esempio '|', ';' e '>', che i metacaratteri jolly. Permettono inoltre di considerare come un unico argomento ciò che è contenuto all'interno degli apici. Questo fornisce la possibilità di salvare stringhe, che contengono metacaratteri ed altri caratteri particolari come lo spazio, all'interno di variabili: funzionalità che riveste una certa utilità se sfruttata con il comando 'eval' (si veda il paragrafo 4.3).

La differenza principale tra i due tipi di apici riguarda l'inibizione o meno del valore delle variabili: questa è possibile solo attraverso gli apici singoli. Ad esempio:

```
Comando> var="value"
```

```
Comando> echo "$var"
```

```
value
```

```
Comando> echo '$var'
```

```
$var
```

Se un argomento viene interamente racchiuso da apici, prima che sia salvato vengono rimossi gli apici. Visto che l'espansione dei metacaratteri jolly e la sostituzione di variabili con il loro valore avviene solo dopo il parsing di un argomento, si utilizzano due variabili globali per determinare se un argomento deve essere modificato o meno: se era eventualmente racchiuso fra apici, l'informazione sarebbe altrimenti persa causa la rimozione degli apici stessi. Le due variabili globali - 'to_match' e 'to_invoke' - sono impostate a 1 all'inizio dell'analisi di un nuovo argomento. Se questo si rivela essere racchiuso tra apici doppi, la sola variabile 'to_match' viene impostata a 0 - segnalando dunque che l'argomento non deve essere sottoposto all'espansione dei percorsi. Se è racchiuso tra apici singoli,

[3]Se a *bash* viene infatti fornito un argomento che presenta una sola occorrenza di apici singoli o doppi non inibiti, continua a chiedere input da parte dell'utente da intendersi come la continuazione dell'argomento precedente. La richiesta di input termina quando viene riscontrata una seconda occorrenza della tipologia di apice non chiuso.

entrambe vengono impostate a 0.

Ovviamente, gli algoritmi di espansione e di sostituzione inibiscono automaticamente anche parti dell'argomento racchiuse fra apici. Ad esempio:

```
Comando> var="value"
Comando> echo test-'$var'-$var
test-$var-value
Comando> echo test-"*"
test-*
```

Infine, a differenza di quanto avviene in *bash* o *bourne*, per semplicità è stata deciso di gestire il caso di apici innestati - anche di diverso tipo - in *smallsh* inibendone il contenuto e successivamente rimuovendoli. Se, infatti, sia *bourne* che *bash* gestiscono il seguente comando:

```
$ echo "test-'*'-test"
test-'*'-test
```

In *smallsh*, invece, il comando ha esito:

```
Comando> "test-'*'-test"
test-*-test
```

4.3 Comando 'eval'

Il comando built-in 'eval' non era presente nella versione originale di *smallsh*. Il comando prende in input uno o più parametri, esegue alcune operazioni su essi, e infine li interpreta come se fossero un input fornito da linea di comando.

Le operazioni che il comando compie sugli argomenti sono principalmente tre: la rimozione degli spazi, la sostituzione del valore di variabili e l'espansione dei metacaratteri jolly. La rimozione degli spazi prevede la divisione in due o più argomenti in presenza del carattere spazio. Ad esempio, il singolo argomento "ls -l", contenente uno spazio, viene scisso nei due argomenti 'ls' e '-l'. Le altre due operazioni sono completamente analoghe - sfruttano, infatti, le stesse funzioni - a quanto visto rispettivamente nei capitoli 3 e 4. Si noti che la sostituzione di variabili non modifica il numero totale di argomenti, a differenza dell'eventuale espansione dei metacaratteri jolly.

Queste operazioni sono svolte in ordine, considerando gli argomenti uno alla volta, e

terminano quando non sono più presenti argomenti.

Il comando permette di eseguire comandi elaborati, ad esempio:

```
Comando> v1="echo ls -l"; v2="-a"; v3='$v1 $v2 '
Comando> eval $v3
ls -l -a
```

In questo caso, *smallsh* sostituisce in autonomia '$v3' con il suo valore. I passi che il comando esegue sono, nell'ordine: separazione degli argomenti '$v1' e '$v2'; sostituzione di '$v1' con il suo valore; separazione degli argomenti 'echo', 'ls' e '-l'; sostituzione di '$v2' con il suo valore. Alla fine si ottengono i quattro argomenti "echo ls -l -a", che vengono poi considerati come un unico comando, il quale viene poi eseguito.

4.4 Esecuzione di script

La versione originale di *smallsh* permetteva già di eseguire script di altre shell grazie alla primitiva 'execvp()' - a cui viene fornito il nome dello script con permesso d'esecuzione e gli eventuali parametri.

Al fine di assumere un comportamento simile a quello di *bash*, si è stabilito di consentire l'esecuzione di uno script anche passando all'invocazione di *smallsh* il nome dello script come primo parametro (dopo l'eventuale presenza di un'opzione) [1]. A differenza di *bash*, è richiesto che lo script abbia il permesso d'esecuzione. Eventuali ulteriori parametri dopo il nome dello script sono da intendersi come parametri con cui invocare lo script stesso. Con questa modalità di invocazione, *smallsh* svolge un ruolo di 'tramite' per l'esecuzione di script, al termine della quale viene interrotto anche il processo di *smallsh*.

Supponendo che il file 'script.sh' sia eseguibile e nella directory corrente, le seguenti invocazioni di *smallsh* portano all'esecuzione dello script:

```
$ smallsh script.sh
$ smallsh -v script.sh par1
```

Nel secondo caso, infatti, si interpreta come nome di script il primo parametro fornito all'invocazione che non sia un'opzione: appunto, 'script.sh'. Sempre nel secondo caso, il parametro 'par1' viene passato ad 'execvp()' - quindi lo script sarà eseguito con quel singolo parametro.

4.5 Espansione dei comandi

Così come presente in *bash* [3], in *smallsh* è stata aggiunta la possibilità di espandere un comando con l'output dello stesso, racchiudendolo fra una coppia di metacaratteri `. Quindi, durante l'analisi dell'input da parte dell'utente, se viene individuato un argomento che contiene una coppia di metacaratteri ‘, si considera il contenuto degli apici rovesci come un comando da eseguire. Sono stati valutati due diversi approcci implementativi: il primo che è stato implementato era basato sull'utilizzo di un file anonimo, successivamente sostituito dal secondo, che utilizza, invece, le pipe. Nel presente paragrafo, saranno esaminate entrambe le implementazioni e si giustificherà la scelta finale di utilizzo delle pipe.

Con il primo approccio implementativo, per prima cosa, si alloca un file anonimo - tramite la funzione di libreria ‘memfd_create()’[4] [10]. In seguito, viene salvata l'eventuale coda dell'argomento - ossia, una parte terminale dell'argomento non compresa tra apici rovesci - e si predispone l'argomento per l'esecuzione: si sostituiscono eventuali spazi con ’/0’ e si salvano i puntatori al primo carattere di ogni sotto-argomento in un apposito buffer. Infine, si invoca la funzione di *smallsh* che si occupa di eseguire i comandi, ossia ‘runcomando()’, fornendo il buffer - con gli argomenti predisposti all'esecuzione - e il descrittore del file anonimo su cui scrivere. A eseguire il comando (si veda il par. 1.3.2) è un processo figlio del processo di *smallsh*. Dopo che il processo padre ha letto - BUF_SIZ byte alla volta - il contenuto della pipe e questa è stata chiusa, si procede con la sostituzione: la parte d'argomento contenuta tra apici rovesci (oppure tutto, se gli apici rovesci lo racchiudono interamente) viene eliminata da ‘tokbuf’ - il buffer con gli argomenti forniti dall'utente - e sostituita con l'esito del comando eseguito dal processo figlio, letto sul file anonimo. Infine, se era presente una coda all'argomento, questa viene riagganciata al termine dell'esito del comando, senza operare alcuna altra modifica su essa.

Il secondo approccio per la gestione dell'esecuzione del comando - il cui problema principale consiste nel recupero dell'esito del comando eseguito - si basa sull'utilizzo di una pipe. Come sopra accennato, prevede la creazione di una pipe, attraveso la funzione

[4]Il metodo ritorna il file descriptor al file anonimo creato.

di libreria 'pipe()'[5]. Dopo aver predisposto, anche in questo caso, la parte del comando racchiusa fra apici rovesci come visto sopra, il buffer con il comando da eseguire viene fornito alla funzione di *smallsh* 'runcomando()', assieme al lato della pipe da chiudere prima dell'esecuzione e il lato della pipe su cui scrivere. In questo caso, si noti, vi è la necessità di fornire un file descriptor da chiudere: il lato di scrittura di una pipe deve essere chiuso prima di poter leggere. Ovviamente, sarà il processo figlio a chiudere il lato di lettura della pipe: il processo padre, infatti, lo dovrà mantenere aperto per poter recuperare l'esito del comando. Leggendo successivamente dalla pipe, si può recuperare l'output del comando eseguito, per poi procedere a sostituirlo in 'tokbuf'.

Tra i due approcci, è stato inizialmente implementato quello basato su file anonimi; successivamente è stato sostituito con il secondo descritto, per alcuni motivi. In primo luogo, la funzione di libreria 'memfd_create()' non ha come scopo primario la comunicazione tra due processi, a differenza delle pipe. In secondo luogo, nel caso particolare di output di comandi estremamente elevato, un file temporaneo - a meno che non si utilizzino funzioni di *file-sealing* - non pone un limite ai byte che si possono scrivere; le pipe, invece, risultano bloccanti e limitano automaticamente i dati in eccesso, senza perderli. Infine, è stato determinato che anche *bourne* facesse affidamento sulle pipe per l'implementazione di questa funzionalità [30].

4.6 Stampa del prompt

Nella versione originale di *smallsh*, dopo le prime istruzioni di inizializzazione delle variabili, la shell stampava sullo standard output il prompt "Comando>" ed attendeva un input da parte dell'utente (si veda par. 1.1.3).

Questa sequenza di operazioni iniziali - in particolar modo, la stampa del prompt su standard output - non dipendeva da eventuali redirezioni applicate al programma di *smallsh*. Quindi, a fronte di un'invocazione come:

```
$ smallsh < command.sh > output.txt
```

supponendo che il file 'command.sh' contenga unicamente la linea:

```
echo prova
```

[5]Il metodo richiede come unico parametro un int[2] - in cui saranno impostati i file descriptor per lettura e scrittura della pipe - e ritorna 0 in caso di successo, -1 in caso di insuccesso.

allora, il contenuto del file “output.txt” risultava:

```
prova
Comando> Comando>
```

In primo luogo, si nota la presenza della stringa che funge da prompt: dato che lo standard output dell’esecuzione di *smallsh* era stato forzato al file “output.txt” mediante la redirezione, il prompt viene scritto all’interno del file. Il fatto che compaia due volte è dovuto all’interpretazione dei comandi: al termine della linea di *echo*, *smallsh* stampa un secondo prompt e inizia la lettura dell’input; ricevendo un EOF, interrompe l’esecuzione. Si può poi notare il fatto che l’ordine delle stampe sia invertito: il file contiene prima l’esito del comando, e solo dopo la stringa di prompt. Questo perché quest’ultima non viene stampata con un carattere ‘\n’ finale (si desidera, infatti, che l’utente fornisca il comando sulla stessa linea del prompt). La stampa avviene, allora, solo al flush del buffer di output, ossia al termine dell’esecuzione del processo di *smallsh* [11].

Sia *bourne* che *bash*, in caso di invocazioni delle shell in redirezione, presentano un comportamento diverso da *smallsh*, ma uguale fra loro. Nello specifico, nel caso in cui l’input sia redirezionato - indipendentemente da un’eventuale redirezione dell’output - il prompt non viene stampato. Se invece non si ha una redirezione in input, allora *bash* e *bourne* stampano il prompt sullo standard error [31] - il quale, di solito, è legato al terminale corrente.

Si è deciso di emulare lo stesso comportamento anche in *smallsh*. Ogni volta che deve essere stampato il prompt in attesa di un comando da parte dell’utente, si esegue la seguente porzione di codice:

```
if(isatty(2) && isatty(0))
        fprintf(stderr, "%s ", p);
```

dove *p* è la stringa di prompt e *stderr* è un ‘FILE*’ - fornito dal sistema che - punta allo standard error [16]. La funzione ‘isatty()’ richiede in input un file descriptor; ritorna ‘1’ se il terminale corrente è associato al file passato, ‘0’ altrimenti [22].

Il costrutto condizionale, quindi, controlla che lo standard error e lo standard input siano collegati ad un terminale: in questo caso, il prompt viene stampato. In tutti gli altri casi non si procede alla stampa della stringa. Questa scelta implementativa - che come accennato sopra, rispecchia quelle di *bash* e *bourne* - prevede la possibilità, da parte

dell'utente, di non visualizzare il prompt semplicemente redirezionando lo standard error.

Con l'aggiunta di questa porzione di codice nella nuova versione di *smallsh*, l'invocazione esaminata precedentemente:

```
$ smallsh < command.sh > output.txt
```

farà sì che il contenuto del file 'output.txt' sia unicamente:

```
prova
```

appunto perché lo standard input non è legato ad alcun terminale. Per poi evitare la stampa del prompt a video senza redirezionare in input *smallsh*, è sufficiente redirezionare lo standard error:

```
$ smallsh 2> /dev/null
echo prova
prova
```

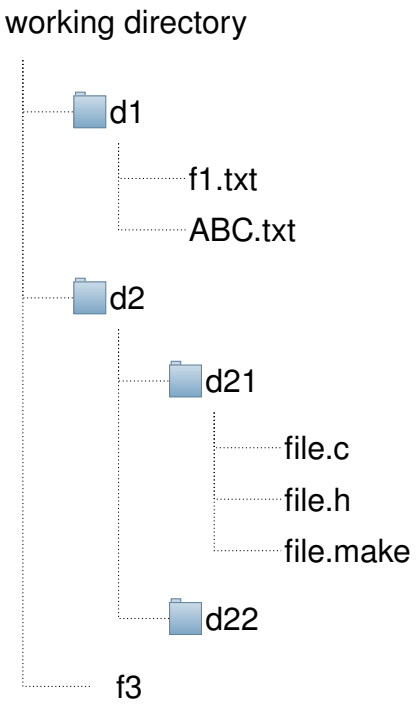
Nonostante l'assenza del prompt, infatti, *smallsh* continua ad attendere e processare i comandi forniti dall'utente.

5 Esempi d'uso

Nel presente capitolo verranno presentati alcuni esempi di utilizzo di *smallsh*. Saranno specificati gli input forniti, verranno evidenziati gli output del programma ed eventualmente approfondite le procedure che sono eseguite da *smallsh*.

5.1 Espansione dei percorsi

Per il presente paragrafo (che si riferisce all'intero capitolo 2), si supponga che la directory di lavoro del processo di *smallsh* sia la seguente:



5.1.1 Il metacarattere '*'

Si considerino i seguenti esempi dell'espansione di percorsi:

```
Comando> echo *
d1 d2 f3
Comando> echo */
d1/ d2/
```

Si ricorda, infatti, che il metacarattere '*' può essere espanso ad una qualsiasi stringa, compresa la stringa vuota. Nell'ultimo caso, vengono mostrate solo le directory: ponendo infatti uno slash alla fine della stringa su cui eseguire il pattern, l'espansione dei percorsi

valuterà solo directory per il livello di innestamento più profondo. Così, infatti:

```
Comando> echo */*/
d2/d21/ d2/d22/
```

Poichè in questo caso sono state richieste le directory del secondo livello di innestamento rispetto alla directory di lavoro. Invece, senza l'uso dello slash finale:

```
Comando> echo */*
d1/f1.txt d1/ABC.txt d2/d21 d2/d22
```

Si noti che, sebbene sia *bash* che *bourne* utilizzino il metacarattere '*' in maniera identica a *smallsh*, solamente *bash* introduce la possibilità di restringere la ricerca a directory utilizzando uno slash terminale nella stringa del pattern.

5.1.2 Il metacarattere '?'

Si consideri quanto segue:

```
Comando> echo */*.txt
d1/f1.txt d1/ABC.txt
Comando> echo */??.txt
d1/f1.txt
```

Negli ultimi due casi, il pattern effettuerà match con i file che abbiano almeno quattro caratteri - di cui gli ultimi quattro devono coincidere con '.txt' - e che siano al secondo livello di innestamento. L'ultimo caso utilizza i metacaratteri '?' - i quali sono espansi ad uno ed un solo carattere - ed aggiunge quindi il vincolo che il numero di caratteri debba essere pari a 6 per i file del secondo livello di innestamento. Nel seguente modo si possono ottenere, per esempio, tutti i file presenti nel terzo livello di innestamento con un'estensione di un solo carattere:

```
Comando> echo */*/*.?
d2/d21/file.c d2/d21/file.h
```

Si noti la differenza con il seguente esempio, in cui l'estensione può essere di qualsiasi lunghezza:

```
Comando> echo */*/*.*
d2/d21/file.c d2/d21/file.h d2/d21/file.make
```

Si ricorda che il metacarattere '?' presenta la stessa funzionalità implementata in *smallsh*

anche nelle shell *bourne* e *bash*.

5.1.3 Elementi nascosti

Come descritto nel paragrafo 2.3.2, eventuali elementi nascosti (e con essi si intendono anche le directory '.' e '..') sono mostrati unicamente se il pattern fornito ha inizio con il carattere '.'. Ad esempio, si ha che:

```
Comando> echo *.
*.
```

Poichè non vi sono elementi nella directory corrente non nascosti che terminano con il carattere '.', quindi viene interpretato il pattern come una semplice stringa. Mentre si ha che:

```
Comando> echo .*
.  ..
```

Si noti che la stessa gestione dell'espansione ad elementi nascosti di *smallsh* è presente anche in *bourne* e *bash*.

5.2 Utilizzo di variabili

Nel presente paragrafo si farà riferimento all'intero capitolo 3. Si supponga per semplicità, se non diversamente indicato, la completa assenza sia di variabili d'ambiente sia di variabili locali, fatto salvo il caso delle variabili speciali (si veda il paragrafo 3.2.7).

5.2.1 Gestione di variabili locali

Si consideri il seguente esempio:

```
Comando> var="value"
```

Il comando crea una variabile locale, di nome 'var' e di valore 'value'. Le seguenti varianti al comando sono identiche all'esempio di cui sopra:

```
Comando> var='value'
Comando> var=value
Comando> var=; var+=value
```

Nel terzo caso, *smallsh* divide il comando in due sotto-comandi: il primo crea la variabile, se non esiste, e le assegna come valore una stringa vuota; il secondo aggiunge in coda

al valore della variabile ‘var’ la stringa ‘value’. In questo modo, il risultato finale è il medesimo.

Al contrario, le seguenti inizializzazioni non andranno a buon fine:

```
Comando> echo var=value
Comando> var = value
```

Per la corretta assegnazione di un valore ad una variabile, infatti, il comando deve essere unicamente composto da una sola stringa che rispetti il formato “nome=[valore]”, con il campo ‘valore’ eventualmente coincidente con la stringa vuota.

Si osservi l’esempio seguente:

```
Comando> var=value
Comando> echo $var
value
```

Il metacarattere ‘$’ viene utilizzato per sostituire al nome di una variabile il suo valore.

Se una variabile non esiste, la sostituzione avviene ugualmente con la stringa vuota:

```
Comando> echo "Valore:  $var2"
Valore:
```

Il metacarattere ‘$’ può essere inibito attraverso l’uso degli apici singoli. Si veda, ad esempio:

```
Comando> var=value
Comando> echo 'Valore:  $var'
Valore:  $var
```

Per la rimozione di una variabile locale, è sufficiente utilizzare il comando built-in ‘unset’, specificando il nome della variabile da eliminare:

```
Comando> var=value
Comando> echo "Valore:  $var"
Valore:  value
Comando> unset var
Comando> echo "Valore:  $var"
Valore:
```

All’inizio, viene creata la variabile ‘var’ e le viene assegnata il valore ‘value’. Successivamente, essa viene eliminata dall’insieme delle variabili locali.

Le operazioni che riguardano le variabili locali - l'inizializzazione, la modifica, la visualizzazione e la rimozione - sono condivise, nel comportamento e nella sintassi, sia da *bourne* che da *bash*. L'unico elemento che differisce riguarda la modifica in append, mediante la sintassi '+=': questa è presente solamente in *bash*, ma non in *bourne*.

5.2.2 Gestione di variabili d'ambiente

Le variabili locali non vengono ereditate dai processi figli. Nel seguente caso, ad esempio:

```
Comando> var=value
Comando> echo "Valore:  $var"
Valore:  value
Comando> smallsh
Comando> echo "Valore:  $var"
Valore:
```

Infatti, il secondo processo di *smallsh* non eredita le variabili locali dal processo che lo genera. Ciò che viene ereditato tra i processi sono le variabili d'ambiente. Si possono promuovere le variabili locali a variabili d'ambiente mediante il comando built-in 'export', a cui deve essere fornito solo il nome della variabile da promuovere:

```
Comando> var=value
Comando> export var
Comando> smallsh
Comando> echo "Valore:  $var"
Valore:  value
```

La modifica - anche in 'append' - e la visualizzazione di variabili d'ambiente sfrutta la stessa sintassi delle variabili locali. Quindi, ad esempio:

```
Comando> var=value
Comando> export var
Comando> var+=1
Comando> echo "Valore:  $var"
Valore:  value1
```

Pur mantenendone il suo stato di variabile d'ambiente.

Il comando 'export' di *smallsh* è presente, con uguale sintassi e funzionalità, sia in *bash* che in *bourne*.

5.3 Comando ‘cd’

Il comando built-in ‘cd’, già presente nella versione originale di *smallsh* - poi ulteriormente sviluppato (come descritto nel par. 4.1) - permette di modificare la directory di lavoro corrente fornendone il path relativo o assoluto.

Per mostrare gli esempi di funzionamento del comando, si supponga che il percorso corrente sia ‘/home/utente’. Supponendo che esista, nella directory corrente, una directory chiamata ‘d1’, è possibile spostarsi al suo interno fornendo un percorso assoluto:

```
Comando> cd /home/utente/d1
Comando> pwd
/home/utente/d1
```

Oppure, utilizzando un semplice percorso relativo:

```
Comando> cd d1
Comando> pwd
/home/utente/d1
```

Nel caso in cui la directory o il percorso fornito non esistano, *smallsh* restituisce un errore. Supponendo che non esista una directory chiamata ‘d3’ all’interno del percorso corrente, i seguenti comandi saranno errati:

```
Comando> cd d3
d3:  No such file or directory
Comando> cd /home/utente/d3
/home/utente/d3:  No such file or directory
Comando> cd d3/d4
d3/d4:  No such file or directory
```

L’invocazione del comando ‘cd’ può avvenire senza parametri; nel caso, viene sottinteso che lo spostamento di directory deve avvenire verso il percorso contenuto nella variabile d’ambiente ‘HOME’. Quindi si ottiene questo comportamento:

```
Comando> cd
Comando> pwd
/home/utente
```

ma solo nel caso che la variabile ‘HOME’ contenga la stringa “/home/utente”. Nel caso particolare in cui la variabile non esista - magari perchè rimossa con il comando built-in

‘unset’ - si viene informati tramite la stampa di un errore:

```
Comando> cd
Comando> smallsh:  cd:  HOME not set
```

Le funzionalità aggiunte nella nuova versione di *smallsh* circa il comando ‘cd’ sono presenti in maniera identica anche in *bourne* e in *bash*. Entrambe le shell suppongono che - se viene eseguito il comando ‘cd’ in assenza di parametri - lo spostamento deve avvenire verso la HOME directory; oltretutto, entrambe le shell, nel raro caso in cui non esista la variabile d’ambiente HOME, stampano un errore apposito.

5.4 Inibizione

L’espansione dei percorsi e la sostituzione di variabili possono essere inibiti mediante caratteri particolari in *smallsh*, come visto nel par. 4.2. All’interno di questo paragrafo, si supponga che sia presente in *smallsh* una variabile locale di nome ‘var’ e valore “value”.

Il carattere degli apici doppi viene utilizzato per inibire solamente l’espansione dei percorsi, mentre il carattere dell’apice singolo è usato anche per l’inibizione della sostituzione di variabili. Ad esempio, infatti:

```
Comando> echo "*"
*
Comando> echo "$var"
value
Comando> echo '$var'
$var
```

Come si può notare anche nei successivi esempi, i caratteri inibitori sono rimossi prima che il comando venga interpretato - anche se questi si trovano in mezzo all’argomento:

```
Comando> echo test-"*"-test
test-*-test
```

Nel caso in cui una tipologia di carattere inibitore compaia in numero dispari - ossia, una sua occorrenza rimanga spaiata - *smallsh* suppone che il carattere mancante sia da porre al termine del comando:

```
Comando> echo test-"*
test-*
```

```
Comando> echo '$var
$var
```

Perchè, appunto, i due comandi vengono interpretati come se la seconda occorrenza mancante del carattere inibitore fosse presente al termine del comando. Oltretutto, i caratteri inibitori permettono di segnalare a *smallsh* la presenza di un singolo argomento, inibendo dunque anche i metacaratteri. Quindi, anche nel seguente caso:

```
Comando> echo "test; echo test2&"
test; echo test2&
```

In questo modo, quanto è compreso fra la coppia dei caratteri inibitori viene ritenuto come un unico argomento. Questa implementazione permette, oltretutto, di poter salvare, come valori di variabili, delle stringhe con i caratteri di spazio e i metacaratteri:

```
Comando> var2="echo test"
Comando> echo $var2
echo test
```

Questa funzionalità si rivela utile per l'utilizzo del comando 'eval' (si vedano i paragrafi 5.3 e 6.5).

Infine, all'interno dello stesso argomento è possibile concatenare l'uso dei caratteri inibitori:

```
Comando> echo "*-'$var'-*-$var"
*-$var-*-value
```

L'esito, appunto, è dovuto al fatto che *smallsh* considera ogni sotto-stringa compresa fra apici a sé stante; dopo averla inibita, gli apici sono rimossi.

Il comportamento delle due tipologie di apici implementata in *smallsh* coincide con quanto avviene nelle shell *bourne* e *bash*. Una differenza sostanziale, però, tra le shell riguarda la gestione del caso in cui compaiono apici spaiati: in *smallsh*, come si è visto, si suppone semplicemente che l'apice mancante sia da porre al termine dell'input fornito dall'utente. In *bourne* e in *bash*, al contrario, viene richiesto all'utente input ulteriore finché non viene riscontrata un'occorrenza dell'apice cercato. Un'ulteriore differenza consistente riguarda il caso di apici innestati. Se *smallsh* semplicemente inibisce il contenuto e poi rimuove le coppie di apici, in *bash* e *bourne* il comportamento è diverso e si basa sul fatto che gli apici doppi inibiscano anche gli apici singoli [4]. Questa particolare scelta implementativa non è presente in *smallsh*.

5.5 Comando ‘eval’

Il comando built-in ‘eval’ permette di ‘valutare’ gli argomenti che vengono forniti, per poi eseguirli. Come visto nella descrizione della sua implementazione (par. 4.3), il comando compie le seguenti operazioni in ordine sugli argomenti passati: rimuove eventuali spazi, effettua la sostituzione di variabili, espande i metacaratteri jolly e rimuove nuovamente, se presenti, i caratteri di spazio. Terminate le operazioni, interpreta gli argomenti forniti come se fossero un input da parte dell’utente.

Il comando ‘eval’ suppone che eventuali spazi separino due argomenti distinti:

```
Comando> "echo test"
echo test:  No such file or directory
Comando> eval "echo test"
test
```

Ricordando che infatti i caratteri inibitori permettono di fornire un unico argomento, anche contenente spazi e metacaratteri, si ha che la prima esecuzione non va a buon fine: *smallsh* tenta di trovare l’eseguibile chiamato ‘echo test’, che in questo caso non esiste. Nel secondo caso, il comando ‘eval’ separa l’argomento fornito in due argomenti distinti - ‘echo’ e ‘test’ - i quali vengono correttamente eseguiti.

Dopo questa prima fase, il comando esegue la sostituzione delle variabili:

```
Comando> var=value
Comando> echo '$var'
$var
Comando> eval "echo '$var'"
value
```

In questo caso, ‘eval’ separa, per prima cosa, l’unico argomento fornito in due argomenti separati. Poi svolge la sostituzione di variabili: in questo modo, i due argomenti diventano “echo” e “value”. Infine, non essendoci più possibilità di svolgere operazioni, i due argomenti vengono trattati come un comando: viene così stampata a video la stringa “value”.

Infine, il comando esegue l’espansione dei metacaratteri jolly. Supponendo di trovarsi in una directory vuota:

```
Comando> var="*"
```

```
Comando> echo ".$var"
.*
Comando> eval echo ".$var"
.  ..
```

Infatti, il comando eval prima espande la variabile di nome ‘var’ al suo valore; poi, espande la stringa “.*” contenente il metacarattere ‘*’, ossia il valore della variabile sostituita. Essendo appunto la directory vuota, gli unici match avvengono con i riferimenti ‘.’ e ‘..’.

Lo stesso comportamento del comando built-in ‘eval’ analizzato in questo sottoparagrafo è presente nelle shell *bourne* e *bash*.

5.6 Esecuzione di script

Come si è visto nel paragrafo 4.4, *smallsh* ora interpreta eventuali argomenti - oltre le opzioni - come se fossero uno script eseguibile ed i parametri con cui eseguirlo.

Supponendo che esista uno script chiamato ‘script.sh’ nella directory corrente ma che sia senza diritto d’esecuzione, tentando di eseguirlo tramite *smallsh* si ottiene il seguente errore a video:

```
$ smallsh script.sh
script.sh:  Permission denied
```

L’errore sopra riportato si verifica provando ad invocare *smallsh* sia da *smallsh* stessa che da *bash*.

Fornendo invece il permesso d’esecuzione a ‘script.sh’ e supponendo che il suo codice stampi semplicemente i parametri forniti in questo modo:

```
#!/bin/bash
echo "Parametri: $*"
```

Allora gli output dello script saranno, nei seguenti casi:

```
$ smallsh script.sh
Parametri:
$ smallsh script.sh 1 2
Parametri:  1 2
```

Perché, infatti, eventuali altri argomenti oltre il primo - che viene inteso come nome di un

eseguibile - saranno forniti come parametri all'esecuzione.
La possibilità di eseguire uno script come parametro di invocazione di una shell è presente sia in *bourne* che in *bash*. Si noti però che, a differenza di *smallsh*, né *bourne* né *bash* necessitano del permesso di esecuzione sul file di script che devono eseguire.

5.7 Espansione dei comandi

Il presente paragrafo si riferisce alla funzionalità dell'espansione dei comandi, la cui implementazione è già stata descritta nel paragrafo 4.5.

Si è vista l'implementazione degli apici rovesci (`) per l'espansione dei comandi. Ad esempio, il seguente comando permette di visualizzare - con una sola linea di input - il path assoluto di un'eventuale directory di nome 'd1' all'interno del path corrente:

```
Comando> echo `pwd`/d1
/home/utente/d1
```

Ovviamente supponendo, in questo esempio, che il percorso corrente sia '/home/utente'. Il comando 'pwd' viene espanso al suo output - ossia, al working path corrente. Quindi, *smallsh* si troverà a processare il comando 'mkdir' seguito dall'intero percorso assoluto della directory 'd1' da creare.

L'espansione dei comandi può risultare utile per poter salvare all'interno di variabili l'esito di determinate operazioni. Ad esempio, per immagazzinare il percorso della directory di lavoro corrente in una variabile locale:

```
Comando> mypath=`pwd`
Comando> echo $mypath
/home/utente
```

Sempre ipotizzando di trovarsi nel percorso '/home/utente'. In questo modo, il risultato del comando verrà mantenuto per l'intera durata del processo di *smallsh* all'interno di una variabile locale chiamata 'mypath', anche se si dovessero effettuare dei cambi di directory.
La possibilità di espandere i comandi mediante gli apici rovesci è presente, con sintassi identica, sia in *bash* che in *bourne*.

5.8 Stampa del prompt

Il presente sottoparagrafo esaminerà il comportamento di *smallsh*, nel caso cui sia invocato con redirezione dell'input, dell'output e dell'error (si veda il par. 4.6).
Si supponga, come esempio iniziale, di invocare senza redirezione *smallsh* e di avviare l'esecuzione di un semplice comando:

```
$ smallsh
Comando> echo prova
prova
Comando>
```

Si può notare, in primo luogo, come sia il prompt - la stringa "Comando>" - che l'esito del comando - la stringa "prova" - siano stati stampati sul terminale, poiché coincidente con lo standard output (si veda il par. 1.1.2). Si supponga invece di redirezionare lo standard output verso un file chiamato "output.txt" e di eseguire lo stesso comando di cui sopra; il comportamento, in questo caso, è il seguente:

```
$ smallsh > output.txt
Comando> echo prova
Comando>
```

Si nota immediatamente che l'esito del comando non viene riportato sul terminale; si troverà invece sul file "output.txt", che conterrà appunto la sola linea:

```
prova
```

Ovviamente, per visualizzare il contenuto del file "output.txt" è necessario uscire da *smallsh* dato che si è attuata la ridirezione dello standard output. Nel caso in cui si redirezioni lo standard error, invece, il prompt non verrà visualizzato, ma l'attesa dei comandi e la loro esecuzione saranno invariati:

```
$ smallsh 2> /dev/null
echo prova
prova
```

Infine, nel caso in cui *smallsh* sia invocata con redirezione in ingresso, indipendentemente dalla redirezione in output, il prompt non sarà stampato.
La gestione della stampa del prompt in funzione della redirezione in input o in output all'invocazione di *smallsh* è identica nel comportamento sia a *bourne* che a *bash*.

6 Conclusioni

Il lavoro di implementazione delle nuove funzionalità di *smallsh*, descritte nel presente elaborato, è stato significativo per approfondire la teoria e le applicazioni della programmazione di sistema. Nello specifico, non solo circa il funzionamento di diverse shell, ma anche l'uso di funzioni di libreria particolarmente avanzate, disponibili sui sistemi operativi UNIX.

Nello specifico, si è rivelata una fonte di sperimentazione stimolante l'implementazione dell'espansione dei comandi: infatti, a differenza di altre funzionalità aggiunte, si è differenziata per lo sviluppo, l'analisi e la scelta tra due implementazioni diverse. Similmente è stato il caso della funzionalità di stampa del prompt, in cui è stata necessaria un'analisi del codice di altre shell per l'adozione di comportamenti analoghi per *smallsh*. Superata la difficoltà iniziale dovuta alle sintassi articolate dei codici sorgente, si è scoperto d'estremo interessi lo studio dell'implementazione della stampa del prompt nelle altre shell.

Pur non presentando funzionalità innovative rispetto ad altre shell - tra cui quelle che hanno ispirato il lavoro svolto, *bash* e *bourne* - il codice sorgente di *smallsh* si presterebbe ad ulteriori estensioni. Fra queste si suggeriscono, ad esempio, la definizione di funzioni di shell, per permettere maggiore flessibilità nell'utilizzo di comandi anche complessi. Un'ulteriore funzionalità non implementata al momento è la cronologia degli input, grazie a cui è possibile ripetere l'esecuzione di comandi precedenti, facilitando il lavoro di un utente. Oppure ancora, l'interpretazione interna di file di script, evitando di delegare l'esecuzione ad altri programmi di shell - rendendo *smallsh* più indipendente.

Per concludere, il lavoro svolto su *smallsh* non ha portato ad uno strumento più efficiente o più funzionale di altre soluzioni già presenti. Ha però permesso di approfondire alcune tematiche sul funzionamento dei sistemi UNIX e sullo sviluppo di software complesso.

Appendici

A Appendice A: La libreria ‘dirent.h’

Per l’implementazione dell’espansione dei metacaratteri jolly - come si è visto nel capitolo 3 - si è rivelato necessario l’utilizzo della libreria ‘dirent.h’ [12]. Grazie alle funzioni che definisce, infatti, è possibile recuperare il nome e la tipologia di file contenute in uno specifico percorso. La presente appendice ne analizza sommariamente il funzionamento.

A.1 Le struct ‘DIR’ e ‘dirent’

La *struct DIR* rappresenta un flusso di elementi interni ad una directory. In maniera analoga al tipo *FILE*, è possibile ottenere progressivamente gli elementi presenti nello stream. La struct DIR contiene, in particolare, una *struct dirent* [12]: tale struct possiede gli attributi utili a rappresentare un generico file. Nello specifico, contiene le seguenti importanti proprietà:

- **char d_name[]**: il nome del file in forma di stringa.
- **ino_t d_ino**: l’*inode number* del file, ossia un riferimento univoco alla posizione del file in memoria. Il datatype ‘ino_t’ è definito nell’header ‘sys/types.h’, e coincide con un unsigned integer per i sistemi UNIX.
- **unsigned char d_type**: la tipologia del file. Sono definite alcune costanti per indicarne il significato; fra queste, *DT_DIR* se si tratta di una directory, o *DT_REG* se si tratta di un file regolare.

La documentazione sottolinea come le struct *DIR* e *dirent* non debbano essere allocate in modo diretto. Diverse funzioni della libreria ‘dirent.h’, infatti, restituiscono dei riferimenti già allocati e inizializzati a queste strutture.

A.2 Apertura di directory

Per aprire - e successivamente esaminare - i contenuti di una directory, si utilizza la funzione [14]:

```
DIR *opendir(const char *dirname);
```

Il parametro ‘dirname’ indica il nome della directory da aprire. Il valore di ritorno è un puntatore ad uno stream dei file contenuti nella directory aperta. In caso di errore, la funzione ritorna un puntatore a NULL.

Un esempio di utilizzo si trova all’interno della funzione ‘search_and_save()’, descritta nel paragrafo 2.3.2. Viene infatti aperta la directory il cui path corrisponde al pattern fornito, fino ad un certo grado di profondità. Una volta aperta, viene fornita ricorsivamente alla funzione, che ne esamina il contenuto e procede nella ricerca di match con il pattern. La sezione di codice interessata è:

```
d2 = opendir(possible_path);
if(d2 == NULL){
        printf(``smallsh: error in opening directory\n");
        path_index -= (strlen(dir->d_name) + 1);
        continue;
}
```

La stringa ‘possible_path’ contiene il percorso della directory da aprire. Se il metodo ‘opendir()’ fallisce ritorna un puntatore a NULL. Nel caso, viene stampata a video una stringa che informa dell’errore e si reimposta il valore di ‘path_index’, che tiene traccia della lunghezza totale del percorso trovato.

A.3 Lettura dello stream

Per scorrere gli elementi dello stream, si utilizza la funzione [15]:

```
struct dirent *readdir(DIR *dirp);
```

Fornito il flusso di elementi attualmente in uso (‘dirp’), la funzione ritorna un puntatore a una *struct dirent*, che rappresenta l’elemento - file o directory - successivo all’interno dello stream. Quando non sono più presenti elementi nello stream, la funzione ritorna un puntatore a NULL.

Ad esempio, per scorrere gli elementi di una directory, al fine di verificare la presenza di match, la funzione ‘search_and_save()’ fa uso della primitiva nel seguente modo:

```
while( (dir = readdir(d)) != NULL ){
        ...
}
```

Ad ogni iterazione, la variabile ‘dir’ di tipo *struct dirent** punta al successivo elemento dello stream *DIR* d* interno alla directory aperta. Se dopo l’assegnamento ‘dir’ non punta a NULL - ovvero, lo stream non è ancora terminato - si utilizza il puntatore per avere informazioni sull’elemento descritto da ‘dir’. Altrimenti, il ciclo termina poichè non sono più presenti elementi da analizzare nello stream di directory.

A.4 Chiusura di directory

La documentazione della libreria [14] elenca gli errori che possono verificarsi mantenendo aperti troppi stream: per evitarlo, i flussi devono essere chiusi. Questo serve a prevenire *memory leaks* dovuti alla mancata deallocazione di risorse non più in uso. La funzione che gestisce la chiusura di directory e il recupero delle risorse allocate è [13]:

```
int closedir(DIR *dirp);
```

Il parametro ‘dirp’ indica la *struct DIR* da chiudere. Il valore di ritorno è pari a ‘0’ in caso di successo, ‘-1’ in caso di fallimento.

A.5 Tipologie di dirent

In alcuni casi, risulta utile esaminare la tipologia del file a disposizione. Per fare ciò, si usa l’attributo *d_type* [9] della *struct dirent* in relazione alle costanti (già accennate nel paragrafo A.1).

Ad esempio, durante la ricerca ricorsiva di percorsi che soddisfino il pattern fornito dall’utente - per l’espansione dei metacaratteri jolly - risulta utile conoscere quali elementi dello stream siano directory o meno. È stata pertanto definita e utilizzata la seguente funzione:

```
int is_directory(unsigned char d_type){
    if(d_type == DT_DIR)        return 1;
    else                        return 0;
}
```

Fornito il ‘d_type’ di una *struct dirent*, la funzione ritorna ‘1’ se esso coincide con la costante *DT_DIR* - ossia, appunto, se è una directory. Altrimenti, ritorna ‘0’.

B Appendice B: Le funzioni per variabili d'ambiente

Per la gestione delle variabili d'ambiente, è stato fatto uso di particolari funzioni contenute nella libreria 'stdlib.h' [17]. Nella presente appendice verranno presentate le tre primitive principalmente utilizzate.

B.1 Recupero valore di variabili d'ambiente

Per ottenere il valore - in formato di stringa - di una variabile di ambiente si sfrutta il metodo [18]:

```
char *getenv(const char *name);
```

Il parametro 'name' richiesto rappresenta il nome identificativo della variabile d'ambiente. Il valore di ritorno è la stringa contenente il valore della variabile richiesta. Se non esiste alcuna variabile d'ambiente con nome 'name', il metodo ritorna un puntatore a NULL.

Nel codice di *smallsh*, il metodo 'getenv()' viene invocato spesso, soprattutto ogni volta che si ha necessità di ottenere il valore di una variabile d'ambiente o per verificare l'esistenza di tale variabile. Ad esempio, si consideri il codice della funzione per ottenere il valore di una generica variabile:

```
char* get_var_value(variable** variables, char* name){

    ...
    char* global_ret;
    if ((global_ret = getenv(name)) != NULL) return global_ret;
    printf("smallsh: variables error: variable \%s not found", name);
    return NULL;
}
```

Nella prima parte omessa della funzione si controlla, ed eventualmente si ritorna, il valore della variabile identificata da 'name' nel caso in cui essa sia locale. Altrimenti, si salva il valore di ritorno di 'getenv()' in 'global_ret': se tale puntatore non è a NULL, allora la primitiva ha trovato la stringa richiesta, e questa viene ritornata. L'ultima linea informa dell'inesistenza della variabile, non avendo trovato riscontri nè localmente nè all'interno dell'ambiente. In teoria, la funzione 'get_var_value()' viene chiamata solo

avendo verificato l'esistenza della variabile con nome 'name': pertanto, l'ultima linea non dovrebbe mai essere eseguita.

B.2 Aggiunta e modifica di variabili d'ambiente

L'inizializzazione e la modifica di variabili d'ambiente avviene mediante il metodo [19]:

```
int setenv(const char *name, const char *value, int overwrite);
```

I parametri rappresentano il nome 'name' della variabile, il suo valore 'value' e un intero attraverso cui imporre la sovrascrittura del valore attuale della variabile, se questa esiste. Se nell'ambiente non è presente una variabile con il nome fornito, allora questa viene creata e le viene assegnato il valore passato come parametro. Se invece la variabile è presente nell'ambiente, il comportamento varia in base al valore dell'intero 'overwrite': se è diverso da 0, il precedente valore della variabile nell'ambiente viene sovrascritto con il valore contenuto nel parametro 'value'; altrimenti, il metodo non esegue alcuna operazione sulla variabile e questa mantiene il suo precedente valore. Il valore di ritorno della primitiva è '0' in caso di successo, '-1' altrimenti.

Nel codice della nuova versione di *smallsh*, è possibile trovare l'utilizzo del metodo 'setenv()' - ad esempio - nella funzione che si occupa di gestire il comando built-in 'export':

```
void export_var(variable** variables, char* name){
    if(!var_exists(variables, name)) return;
    if(is_reserved(variables, name)) return;
    if(getenv(name) != NULL) return;

    if(setenv(name, get_var_value(variables, name), 0) == 0){
        unset_variable(variables, name);
    }
}
```

In primo luogo, la funzione verifica che il nome della variabile passata sia una variabile locale esistente non speciale: con 'var_exists()' si determina che effettivamente esista, con 'is_reserved()' si controlla che non sia speciale e accertandosi che 'getenv()' ritorni NULL si verifica che sia locale. Poi viene invocata proprio la primitiva 'setenv()': passando il

nome della variabile, il suo valore locale ottenuto mediante 'get_var_value()' e 0[6]. Se il metodo ritorna 0 - quindi, è terminata con successo - si invoca la funzione locale 'unset_variable()' per rimuoverla dalle variabili locali.

B.3 Rimozione di variabili d'ambiente

Le variabili d'ambiente, così come sono create, possono essere anche eliminate. Per fare ciò, si sfrutta il metodo [20]:

```
int unsetenv(const char *name);
```

La primitiva richiede il nome 'name' della variabile d'ambiente da eliminare. Il valore di ritorno è pari a 0 in caso di successo, -1 altrimenti. Se la variabile 'name' non esiste nell'ambiente, il metodo non modifica l'ambiente ma termina ugualmente con successo.

Per la gestione delle variabili per *smallsh*, l'uso della primitiva 'unsetenv()' si è rivelato utile per l'implementazione del comando built-in 'unset'. Le operazioni di tale comando sono delegate alla funzione:

```
void unset_variable(variable** variables, char* name){

    ...
    if(getenv(name) != NULL){
        unsetenv(name);
        return;
    }
    printf("smallsh: variables error: no %s variable found\n", name);
}
```

La parte omessa della funzione si occupava di trovare - ed eventualmente rimuovere - la variabile con nome 'name' tra le variabili locali. Nella seconda parte si determina se la variabile è d'ambiente: nel caso 'getenv()' ritorni un valore diverso da NULL, viene invocata la primitiva 'unsetenv()' con il nome della variabile da eliminare. L'ultima linea stampa a video un errore nel caso in cui la variabile non sia né locale, né d'ambiente. In teoria, la funzione 'unset_variable()' viene invocata solo dopo aver verificato che la variabile esista: dunque, l'ultima linea non dovrebbe mai essere eseguita.

[6] Si noti che avendo controllato che la variabile non esista nell'ambiente, non sarà mai da sovrascrivere.

C Appendice C: Differenze implementative di bourne, bash e smallsh

Le due versioni di *smallsh*, quella originale e quella corrente, traggono spunto per il comportamento e le funzionalità rispettivamente dalle shell **Bourne Shell** (da qui innanzi, chiamata semplicemente *bourne*), distribuita nel 1978, e **Bash** (acronimo di 'Bourne Again SHell'), distribuita inizialmente nel 1989. Nella presente appendice, verranno evidenziate le principali caratteristiche tratte dall'una e dall'altra shell, sottolineando anche le differenze presenti fra esse [6].

C.1 Redirezione

Le implementazioni della redirezione nelle due shell - seppur più completo e articolato in *bash* - sono sostanzialmente identiche. Entrambe utilizzano delle *struct* apposite - *IOPTR* per *bourne* [27], *REDIRECT* per *bash* [8] - in cui viene impostata la tipologia di redirezione da eseguire, nonchè il nome del file su cui effettuarla.

Entrambe le shell svolgono la redirezione non appena viene trovato un metacarattere '>' o '<': la 'word' o il 'token' successivo viene poi utilizzato per impostare il nome del file su cui svolgere la redirezione. Dato che la versione originale di *smallsh* ha tratto spunto dalle funzionalità di *bourne*, si può notare l'assenza di alcune funzionalità aggiuntive presenti in *bash*. Fra queste, si ricorda l'espansione del nome del file su cui svolgere la redirezione - purché venga fornito un pattern che faccia match con un solo file. Si evidenzia poi la possibilità, in *bash* ma non in *bourne* né in *smallsh*, di redirigere - attraverso la sintassi molto sintetica `&>` - sia lo standard output che lo standard error sullo stesso file.

C.2 Piping di comandi

La *struct IOPTR* di *bourne* viene oltretutto utilizzata per l'implementazione del piping dei comandi. La presenza di due campi di tipo **struct IOPTR* permette di tenere traccia di comandi in piping fra loro: uno dei campi punterà al comando in input, l'altro al comando in output. Vista la struttura concatenata dell'implementazione, i processi che eseguiranno i comandi saranno l'uno figlio dell'altro. L'ultimo comando sarà eseguito da un figlio del processo di *bourne*; il penultimo, da un nipote del processo di *bourne* - figlio del processo che esegue l'ultimo comando - e così via. Al contrario, *bash* sfrutta un array di *REDIRECT* per implementare il piping fra comandi: in questo modo saranno creati

processi figli - fratelli fra loro - del processo originario della shell per eseguire i diversi comandi.

Anche in questo caso è possibile cogliere come l'implementazione della versione originale di *smallsh* si sia basata, per il piping di comandi, su *bourne*. Infatti, anche *smallsh* crea un processo figlio per l'ultimo comando, il quale si occuperà di generare un figlio che esegua il penultimo; così via, fino al termine dei comandi da eseguire.

C.3 Esecuzioni particolari

Nel presente sottoparagrafo si esamina principalmente l'implementazione di *bourne*, per confrontarla con quella della versione originale di *smallsh*.

In *bourne*, il metacarattere ';' - che segnala la presenza di comandi distinti da eseguire in ordine - pone fine, momentaneamente, al parsing dell'input. La lista di 'token' compilata fino a quel punto viene salvata [28]. Poi il processo riprende il parsing del comando fornito dall'utente. In *smallsh*, invece, appena viene riscontrata un'occorrenza del metacarattere ';', viene avviata l'esecuzione del comando analizzato fino a quel punto, delegata ad un processo figlio. Il parsing dell'input riprende immediatamente, interpretando quanto segue il metacarattere ';' come se fosse un nuovo comando fornito dall'utente.

L'esecuzione in background, che l'utente può imporre attraverso il metacarattere '&', viene compiuta in *bourne* attraverso una funzione che genera un processo figlio a cui viene affidata l'esecuzione del comando. Senza attendere il termine del figlio, il processo di *bourne* prosegue nella sua esecuzione. In *smallsh*, l'implementazione dell'esecuzione in background mediante il metacarattere '&' è identica a quella di *bourne*, sopra descritta.

Si possono notare, però, un paio di differenze implementative tra le due shell durante le esecuzioni particolari. La prima che si può riscontrare riguarda il caso di sequenze di comandi. Infatti, *smallsh* esegue immediatamente - all'individuazione del metacarattere ';' - quanto analizzato fino a quel punto, per poi procedere con il parsing dell'input. Al contrario, *bourne* reinvoca ricorsivamente la funzione di parsing, creando poi una catena di comandi, eseguiti in ordine al termine della fase di parsing. In sostanza, a differenza di *smallsh*, la shell *bourne* mantiene distinte le fasi di parsing e di esecuzione. La seconda differenza riguarda il caso di esecuzione in background: *smallsh*, più efficacemente, imposta semplicemente un flag poi controllato dalla funzione 'runcomando()', che si occupa delle esecuzioni. La *bourne*, invece genera una nuova istanza di una *struct* apposita

con il metodo ‘makefork()’ [29]; l’istanza creata viene poi mandata in esecuzione. La differenza tra le due scelte, ovviamente, è da ricercarsi nei diversi modi di salvataggio dei comandi: in *smallsh* gli argomenti sono contenuti nello stesso array ‘tokbuf’, con un ulteriore array che tiene traccia del punto in cui hanno inizio i diversi argomenti. Dall’altro lato, *bourne* utilizza *struct* apposite per rappresentare i comandi.

C.4 Espansione dei percorsi

La funzionalità di espansione dei percorsi di *smallsh*, per quanto implementata seguendo la documentazione di *bash* [5], è pressoché simile anche alla *bourne*.

Tutte e tre le shell implementano i metacaratteri ‘*’ e ‘?’ con semantica uguale; permettono oltretutto di inibire tali metacaratteri sia con gli apici doppi che con gli apici singoli. Inoltre, tutte gestiscono in maniera analoga il match con un carattere ‘.’ all’inizio di un nome di file, il quale deve essere presente esplicitamente nel pattern fornito. A differenza delle due shell più importanti, però, *smallsh* non ha implementato i metacaratteri ‘[]’ [5]. Una piccola differenza presente tra *bash* e *bourne*, sempre riguardanti la stessa coppia di metacaratteri, ha a che vedere con l’esclusione di caratteri: *bourne* richiede la presenza del carattere ‘!’ [24], mentre *bash* permette anche l’uso del carattere ‘^’ [6].

C.5 Sostituzione delle variabili

La possibilità di creare ed eliminare variabili, nonché di sostituire - mediante il metacarattere ‘$’ - il nome con il valore delle variabili stesse affonda la sua sintassi in *bourne*, poi ereditata in maniera identica da *bash* e da *smallsh*.
Sia *bash* che *bourne* presentano sintassi di invocazione particolari, ad esempio:

```
$ echo ${var- string}
```

in cui viene restituito il valore della variabile con nome ‘var’ solo se questa è stata inizializzata; altrimenti, viene stampata la stringa ‘string’ [25]. Queste invocazioni particolari non sono state implementate in *smallsh*.

Dato che la sostituzione di variabili in *smallsh* basa la sua implementazione su quella di *bash*, queste due shell presentano però una sintassi aggiuntiva rispetto a *bourne* [6]:

```
$ var+=string
```

grazie a cui viene modificato il valore della variabile ‘var’, aggiungendo “string” al termine del suo precedente valore.

7 Bibliografia

[1] Free Software Foundation. *Bash Documentation: Bash Invocation (6.1)*. Consultato il 10/05/2024. 2022. URL: https://www.gnu.org/software/bash/manual/html_node/Invoking-Bash.html.

[2] Free Software Foundation. *Bash Documentation: CD command (4.1)*. Consultato il 10/05/2024. 2022. URL: https://www.gnu.org/software/bash/manual/bash.html#index-cd.

[3] Free Software Foundation. *Bash Documentation: Command Substitution (3.5.4)*. Consultato il 10/05/2024. 2022. URL: https://www.gnu.org/software/bash/manual/bash.html#Command-Substitution.

[4] Free Software Foundation. *Bash Documentation: Double Quotes (3.1.2.3)*. Consultato il 29/05/2024. 2022. URL: https://www.gnu.org/software/bash/manual/html_node/Double-Quotes.html.

[5] Free Software Foundation. *Bash Documentation: Filename Expansion (3.5.8)*. Consultato il 9/05/2024. 2022. URL: https://www.gnu.org/software/bash/manual/html_node/Filename-Expansion.html.

[6] Free Software Foundation. *Bash Documentation: Major differences from The Bourne Shell (Appendix B)*. Consultato il 10/05/2024. 2022. URL: https://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents.

[7] Free Software Foundation. *Bash Documentation: Shell Parameters (3.4)*. Consultato il 10/05/2024. 2022. URL: https://www.gnu.org/software/bash/manual/html_node/Shell-Parameters.html.

[8] Free Software Foundation. *Bash: redir.c (line 241)*. Consultato il 10/05/2024. 2021. URL: https://github.com/bminor/bash/blob/master/redir.c#L241.

[9] Free Software Foundation. *The GNU C library: Format of a Directory Entry (14.2.1)*. Consultato il 9/05/2024. 2023. URL: https://www.gnu.org/software/libc/manual/html_node/Directory-Entries.html#index-struct-dirent.

[10] Free Software Foundation. *The GNU C library: Memory-mapped I/O (13.8)*. Consultato il 10/05/2024. 2023. URL: https://www.gnu.org/software/libc/manual/html_node/Memory_002dmapped-I_002fO.html#index-memfd_005fcreate.

[11] Free Software Foundation. *The GNU C library: Normal Termination*. Consultato il 28/05/2024. 2023. URL: https://www.gnu.org/software/libc/manual/html_node/Normal-Termination.html.

[12] The Open Group. *dirent.h*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/basedefs/dirent.h.html.

[13] The Open Group. *dirent.h: closedir()*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/closedir.html.

[14] The Open Group. *dirent.h: opendir()*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/opendir.html.

[15] The Open Group. *dirent.h: readdir()*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/readdir.html.

[16] The Open Group. *stdio.h: stderr*. Consultato il 29/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/stdin.html.

[17] The Open Group. *stdlib.h*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdlib.h.html.

[18] The Open Group. *stdlib.h: getenv()*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/getenv.html.

[19] The Open Group. *stdlib.h: setenv()*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/setenv.html.

[20] The Open Group. *stdlib.h: unsetenv()*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695399/functions/unsetenv.html.

[21] The Open Group. *unistd.h: exec functions*. Consultato il 9/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009604499/functions/exec.html.

[22] The Open Group. *unistd.h: isatty()*. Consultato il 28/05/2024. 2004. URL: https://pubs.opengroup.org/onlinepubs/009695299/functions/isatty.html.

[23] Keith Haviland and Ben Salama. *UNIX system programming*. 1987.

[24] IBM. *Bourne Documentation: Filename Substitution*. Consultato il 10/05/2024. 2023. URL: https://www.ibm.com/docs/en/aix/7.2?topic=shell-file-name-substitution-in-bourne.

[25] IBM. *Bourne Documentation: Shell Parameters*. Consultato il 10/05/2024. 2023. URL: https://www.ibm.com/docs/en/aix/7.2?topic=shell-conditional-substitution-in-bourne.

[26] IBM. *Order of Directory Contents Returned by Calls to readdir()*. Consultato il 10/05/2024. 2019. URL: https://www.ibm.com/support/pages/order-directory-contents-returned-calls-readdir.

[27] The UNIX Heritage Society. *Bourne Shell: cmd.c (line 334)*. Consultato il 10/05/2024. 1979. URL: https://www.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/sh/cmd.c.

[28] The UNIX Heritage Society. *Bourne Shell: cmd.c (line 70)*. Consultato il 10/05/2024. 1979. URL: https://www.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/sh/cmd.c.

[29] The UNIX Heritage Society. *Bourne Shell: cmd.c (line 90)*. Consultato il 10/05/2024. 1979. URL: https://www.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/sh/cmd.c.

[30] The UNIX Heritage Society. *Bourne Shell: macro.c (line 168)*. Consultato il 28/05/2024. 1979. URL: https://www.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/sh/macro.c.

[31] The UNIX Heritage Society. *Bourne Shell: main.c (line 18)*. Consultato il 29/05/2024. 1979. URL: https://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/sh/main.c.