Resolución de los challenges promocionales

Pseudo-variables

El significado de la variable `$?` es el código de salida del último proceso ejecutado por el shell. El código de salida de los procesos ya está siendo almacenada por el esqueleto en la variable `status` global, implementar la variable mágica `$?` solo implica expandirla a lo que indique `status`. Se modifica `expand_environ_var`:

```
static char* expand_environ_var(char* arg) {
        if (arg[0] == '$') {
+           if (arg[1] == '?') {
+                memset(arg, 0, strlen(arg));
+                snprintf(arg, 4, "%d", status);
+                return arg;
+           }
                char* env = getenv(arg + 1);
                if (!env) {  // Expand into empty variable
                        env = "";
                }
                size_t len = strlen(env) + 1;
                arg = realloc(arg, len);
                memset(arg, 0, len);
                strncpy(arg, env, len);
        }
        return arg;
}
```

> Investigar al menos otras dos variables mágicas estándar, y describir su propósito.

Otras variables internas implementadas por shells son `$0`, `$1`, `$2`, etc. que son los parámetros pasados al ejecutable del shell. Son útiles cuando se ejecutan comandos leyendo desde un archivo en vez de manera interactiva. Luego existe `$!` que indica el PID del último proceso en background ejecutado (con `&`).

Tuberías múltiples

La funcionalidad pedida fue implementada de manera sencilla observando que la estructura que almacena el comando de tipo `PIPE` ya parseado se divide en dos subcomandos. Cuando se le pasa al shell un comando con tuberías múltiples, el primer comando termina en la estructura del primer subcomando, y **todos los demás** terminan en el segundo subcomando. De esta manera, se puede detectar la cantidad de subcomandos en la segunda estructura, y de haber más de uno, llamar recursivamente a la función que ejecuta estos subcomandos, hasta que haya en la segunda estructura un único comando almacenado.

Para esto se modificó la función `handle_pipe`, llamada dentro del `switch`, `case PIPE` de la función en `exec.c`

```
        pid_t first;
        if ((first = fork()) == 0) {
                dup2(pipefds[1], STDOUT_FILENO);
                exec_cmd(pipecmd->leftcmd);
        } else {
            waitpid(first, 0, 0);
                dup2(pipefds[0], STDIN_FILENO);
                close(pipefds[1]);
+       if (strstr(pipecmd->rightcmd->scmd, "|") != NULL) {
+           struct cmd* new = parse_line(pipecmd->rightcmd->scmd);
+           handle_pipe(new);
+       } else {
                    exec_cmd(pipecmd->rightcmd);
        }
        }
```

Segundo plano avanzado

La notificación de fin procesos ejecutados en *background* se hizo implementando una función que se ejecuta al principio de todo el shell, en el `main`:

```
void background_process_notify(int sig) {
    if (background_process) {
```

```
        printf("terminado: PID=%d (%s)\n", background_process, background_prompt);
    }
    background_process = 0;
    memset(background_prompt, 0, PRMTLEN);
}

void add_handler() {
    struct sigaction s;
    memset(&s, 0, sizeof(s));
    s.sa_flags = SA_SIGINFO | SA_RESTART;
    s.sa_handler = background_process_notify;
    sigaction(SIGCHLD, &s, NULL);
}

int main(void) {
    add_handler();
    ...
```

Y también, modificando la rutina de ejecución de procesos para registrar las variables `background_process` y `background_prompt` (globales) cuando se ejecute un comando en background:

```
    if (parsed->type == BACK) {
+       strcpy(background_prompt, parsed->scmd);
+       background_process = p;
        print_back_info(parsed);
        free_command(parsed);
        return 0;
    }
```

> Explicar detalladamente cómo se manejó la terminación del mismo.

Se modifica el *handler* de `SIGCHLD`, señal enviada del proceso hijo (los procesos ejecutados) al padre (nuestra shell) cada vez que el hijo finaliza su ejecución, para ejecutar la función `background_process_notify` definida. Esta función va a imprimir por pantalla los valores de las variables globales mencionadas anteriormente, que fueron oportunamente asignadas los valores del proceso hijo ejecutado en background durante la rutina de lanzamiento. Esta implementación sencilla solo permite notificar correctamente la finalización de 1 proceso en ejecución, pero puede generalizarse a una cantidad arbitraria si se mantiene un array de comandos lanzados en segundo plano.

> ¿Por qué es necesario el uso de señales?

Las señales son la única manera de poder enterarnos de *cuando* funciona un proceso hijo en segundo plano. Al no estar haciendo el `wait` explícito de su finalización, en ningún momento el shell para a fijarse si terminó el proceso. Las señales en UNIX son la única manera de notificar al proceso de eventos externos sin que el proceso mismo haga una lectura explícita de alguna variable o algún otro mecanismo. Ya el estándar determina que una señal es enviada al proceso padre cuando un hijo termina, así que es la manera sencilla de implementar la funcionalidad, solo hay que definir el *handler* para la señal correcta.