```c
// ---- exec.c ----

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>

#include "syntax.h"
#include "jobs.h"
#include "intern.h"

void convert_word (char *s)
     /* convertit s qui peut contenir des ' " \
        et les interpretes
     */
{
  char *r, *w;

  if (*s == '\'') {
    for (r = s + 1, w = s; *r; r++, w++)
      *w = *r;
    *(w-1) = '\0';
  } else if (*s == '"') {
    for (r = s + 1, w=s; *r; r++, w++) {
      if (*r == '\\') {
        r++;
        if (*r == 'n')
          *w = '\n';
        else
          *w = *r;
      } else {
        *w = *r;
      }
    }
    *(w - 1) = '\0';
  } else {
    for (r = s, w = s; *r; r++, w++)
      if (*r == '\\') {
        r++;
        if (*r == 'n')
          *w = '\n';
        else
          *w = *r;
      } else {
        *w = *r;
      }
    *w = '\0';
  }
}

char **convert (scomm_t scomm, char **input, char **output)
{
  char **argv;
  int i, argc = 1;
  elt_list_t l;

  *input = *output = "";

  l = scomm.elt_list;
  while (l) {
    convert_word (l->elt.word);
    switch (l->elt.type) {
    case ARG:
      argc++;
      break;
    case IN:
      *input = l->elt.word;
      break;
    case OUT:
      *output = l->elt.word;
      break;
    }
    l = l->next;
  }

  argv = calloc (argc + 1, sizeof *argv);
  convert_word (scomm.name);
  argv[0] = scomm.name;
  argv[argc] = NULL;
  l = scomm.elt_list;
  i = 1;
  while (l) {
    if (l->elt.type == ARG) {
      argv[i] = l->elt.word;
      i++;
    }
    l = l->next;
  }
  return argv;
}

int is_intern (char *name)
{
  return
    !strcmp (name, "jobs") ||
    !strcmp (name, "wait") ||
    !strcmp (name, "exit") ||
    !strcmp (name, "cd");
}


int exec_extern_scomm (scomm_t scomm)
     /* execute une commande simple (externe)
        (attention: ne fait pas de fork)
     */
{
  char **argv, *input, *output;
  int fd_in, fd_out, ret_val;

  argv = convert (scomm, &input, &output);

  if (strcmp (input, "")) {
    if ((fd_in = open (input, O_RDONLY)) < 0) {
      fprintf (stderr, "ddsh: %s: cannot open.\n", input);
      return -1;
    }
    dup2 (fd_in, STDIN_FILENO);
    close (fd_in);
  }

  if (strcmp (output, "")) {
    if ((fd_out = open (output, O_WRONLY | O_CREAT, 0644)) < 0) {
      fprintf (stderr, "ddsh: %s: cannot open.\n", output);
      return -1;
    }
    dup2 (fd_out, STDOUT_FILENO);
    close (fd_out);
  }
```

```c
  ret_val = execvp (argv[0], argv);
  perror (argv[0]);
  free (argv);
  return abs (ret_val);
}


int exec_if_is_intern (scomm_t scomm, int *ret)
     /* execute une commande interne du shell
        renvoie -1 si erreur
        sinon renvoie renvoie 1 ssi scomm est une commande
        interne. Le code de retour est mis dans *ret
      */
{
  char **argv, *input_name, *output_name;
  FILE *fin = stdin, *fout = stdout;

  if (!is_intern (scomm.name)) {
    return 0;
  }

  argv = convert (scomm, &input_name, &output_name);

  if (strcmp (input_name, ""))
    if ((fin = fopen (input_name, "r")) == NULL) {
      perror (input_name);
      free (argv);
      return -1;
    }

  if (strcmp (output_name, ""))
    if ((fout = fopen (output_name, "w")) == NULL) {
      perror (output_name);
      free (argv);
      return -1;
    }

  if (!strcmp (scomm.name, "jobs")) {
    *ret = dd_jobs (fin, fout, argv);
  } else if (!strcmp (scomm.name, "exit")) {
    *ret = dd_exit (fin, fout, argv);
  } else if (!strcmp (scomm.name, "wait")) {
    *ret = dd_wait (fin, fout, argv);
  } else if (!strcmp (scomm.name, "cd")) {
    *ret = dd_cd (fin, fout, argv);
  }
  if (fin != stdin) fclose (fin);
  if (fout != stdout) fclose (fout);
  free (argv);
  return 1;
}

int wait_pipe (pipe_t p)
{
  int r = 0;
  pipe_t l;

  l = p;
  while (l) {
    if (l->pid > 0)
      if (waitpid (l->pid, &r, 0) < 0)
        perror ("wait_pipe waitpid");
    l = l->next;
  }
  return r;
}

int exec_pipe (pipe_t p, int sync)
     // execute un tube (attention: fait des fork)
{
  pipe_t l;
  int last_out; /* descripteur sur le cot\216 lecture du tube de sortie
                   de la commande pr\216c\216dente */
  pid_t pid;
  int son_stat, r;

  if (!p) return 0;
  if (!sync && !can_create_new_job()) {
    fprintf (stderr, "ddsh: Sorry, job table is full.\n");
    return 1;
  }

  l = p;
  last_out = dup (STDIN_FILENO);
  while (l->next) {
    if (is_intern (l->scomm.name)) {
      fprintf (stderr, "ddsh: Cannot pipe intern command %s, ignoring.\n",
               l->scomm.name);
      l->pid = (pid_t) -1;
    } else {
      pipe (l->fd);
      if ((pid = fork ())) { // pere
        dup2 (l->fd[0], last_out);
        close (l->fd[1]);
        close (l->fd[0]);
        l->pid = pid;
      } else { // fils
        dup2 (last_out, STDIN_FILENO);
        close (last_out);
        dup2 (l->fd[1], STDOUT_FILENO);
        close (l->fd[1]);
        close (l->fd[0]);
        exit (exec_extern_scomm (l->scomm));
      }
    }
    l = l->next;
  }

  if (sync) {
    r = exec_if_is_intern (l->scomm, &son_stat);
    if (r < 0)
      return 1;
    if (r) {
      l->pid = (pid_t) -1;
      if (put_in_fg_job (make_job (p), &son_stat) < 0)
        fprintf (stderr, "There is already a fg job!!!!\n");
      return son_stat;
    }
  }

  if ((pid = fork ())) { // pere
    close (last_out);
    l->pid = pid;
    if (sync) {
      if (put_in_fg_job (make_job (p), &son_stat) < 0)
        fprintf (stderr, "There is already a fg job!!!!\n");
      return son_stat;
    } else {
      add_job (stdout, make_job (p));
      return 0;
    }
  } else { // fils
```

```
    dup2 (last_out, STDIN_FILENO);
    close (last_out);
    exit (exec_extern_scomm (l->scomm));
  }
  return son_stat;
}

int exec_comm (comm_t comm)
     // execute une commande complete (fait les fork)
{
  comm_t l = comm;
  int ret_val = 0;

  while (l) {
    ret_val = exec_pipe (l->pipe, l->sync);
    l = l->next;
  }
  return WEXITSTATUS (ret_val);
}
```

```c
// ---- intern.c ----

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <dirent.h>

#include "syntax.h"
#include "jobs.h"
#include "intern.h"

int dd_jobs (FILE *in, FILE *out, char **argv)
{
  update_print_jobs (out);

  return 0;
}


pid_t get_last_pid (job_t job)
{
  if (!job)
    return (pid_t) -2;

  while (job->next)
    job = job->next;

  return job->pid;
}

int dd_wait (FILE *in, FILE *out, char **argv)
{
  pid_t pid;
  int id = -1, ret;

  if (argv[1]) {
    id = atoi (argv[1]);
  } else
    pid = -1;

  if (pid < -1) {
    fprintf (stderr, "ddsh: %d: no such job\n", id);
    return 1;
  }

  wait_job (id, &ret);

  if (id >= 0)
    update_print_if_dead_jobs (out);

  return ret;
}

int dd_cd (FILE *in, FILE *out, char **argv)
{
  char *pathname;

  if (argv[1])
    pathname = argv[1];
  else
    pathname = getenv ("HOME");

  if (chdir (pathname) < 0) {
    perror ("cd");
    return 1;
  }

  return 0;
}

int dd_exit (FILE *in, FILE *out, char **argv)
{
  int ret_val = 0;
  static int warned = 0;

  if (exists_job() && !warned) {
    fprintf (stderr, "You have runnning jobs.\n");
    warned = 1;
    return 1;
  }

  if (argv[1])
    ret_val = atoi (argv[1]);

  kill_jobs (SIGHUP);
  printf ("May the ddsh be with you.\n");
  exit (ret_val);
}
```

```c
// ---- jobs2.c ----

#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>

#include "syntax.h"
#include "jobs.h"

#define N_SIGS 9

job_t job_tab[MAX_JOBS];
int first_free = 1;

typedef struct st_sig_name_t {
  int num;
  char *name;
} sig_name_t;

sig_name_t sig_name_tab [N_SIGS] =
{ { SIGTERM, "SIGTERM" },
  { SIGKILL, "SIGKILL" },
  { SIGCONT, "SIGCONT" },
  { SIGSTOP, "SIGSTOP" },
  { SIGTSTP, "SIGTSTP" },
  { SIGHUP,  "SIGHUP" },
  { SIGSEGV, "SIGSEGV" },
  { SIGUSR1, "SIGUSR1" },
  { SIGUSR2, "SIGUSR2" } };

char *signame (int signum) {
  int i;
  static char def[20];
  for (i = 0; i < N_SIGS; i++)
    if (sig_name_tab[i].num == signum)
      return sig_name_tab[i].name;

  sprintf (def, "SIG %d", signum);
  return def;
}

void init_job_tab (void)
{
  int id;
  for (id = 0; id < MAX_JOBS; id++)
    job_tab[id] = NULL;
  first_free = 1;
}

int can_create_new_job (void)
{
  return first_free < MAX_JOBS;
}

int exists_job (void)
{
  return first_free > 1;
}

job_t make_job (pipe_t pipe)
{
  job_t l, *p;

  p = &l;
  while (pipe) {
    *p = malloc (sizeof **p);
    (*p)->name = calloc (1 + strlen (pipe->scomm.name),
                         sizeof (char));
    strcpy ((*p)->name, pipe->scomm.name);
    (*p)->pid = pipe->pid;
    (*p)->status = RUN;
    (*p)->code = 0;
    p = &((*p)->next);
    pipe = pipe->next;
  }
  *p = NULL;

  return l;
}

void print_summary_job (FILE *out, int id)
{
  job_t l;

  l = job_tab[id];
  fprintf (out, "[%d]", id);
  while (l) {
    fprintf (out, " %d", (int) l->pid);
    l = l->next;
  }
  fprintf (out, "\n");
}

int add_job (FILE *out, job_t job)
{
  if (!can_create_new_job())
    return -1;

  if (!job)
    return 0;

  job_tab[first_free] = job;
  print_summary_job (out, first_free);

  return first_free++;
}


int remove_job (int id)
{
  job_t l, p;

  if (id < 0 || first_free <= id || !job_tab[id])
    return -1;

  l = job_tab[id];
  while (l) {
    p = l;
    l = l->next;
    free (p->name);
    free (p);
  }

  job_tab[id] = NULL;
  if (id == first_free - 1) {
    do first_free--;
    while (first_free > 0 && job_tab[first_free] == NULL);
    first_free++;
  }
```

```
  return 0;
}


int convert_status (int status, job_t l)
{
  if (WIFEXITED (status)) {
    l->status = EXIT;
    l->code = WEXITSTATUS (status);
    return 0;
  }
  if (WIFSTOPPED (status)) {
    l->status = STOP;
    l->code = WSTOPSIG (status);
    return 1;
  }
  if (WIFSIGNALED (status)) {
    l->status = SIG;
    l->code = WTERMSIG (status);
    return 0;
  }
  return 1;
}


int update_proc_status (job_t l)
{
  int st, r;

  if (l->status == EXIT || l->status == SIG)
    return 0;

  if ((r = waitpid (l->pid, &st, WNOHANG | WUNTRACED)) < 0) {
    return 0;
  }

  if (r == 0) {
    l->status = RUN;
    l->code = 0;
    return 1;
  }

  return convert_status (st, l);

}

int wait_job (int id, int *status)
{
  job_t l;
  int r;

  if (id < 0 || MAX_JOBS <= id || !job_tab[id])
    return -1;

  l = job_tab[id];
  while (l) {
    if (l->pid != (pid_t) -1 &&
        (l->status == RUN || l->status == STOP)) {
      if ((r = waitpid (l->pid, status, 0)) < 0) {
        perror ("waitpid in wait_job");
        return -1;
      } else if (r >= 0)
        convert_status (*status, l);
    }
    l = l->next;
  }
  remove_job (id);
  return 0;
}

int put_in_fg_job (job_t job, int *status)
{
  if (job_tab[0])
    return -1;

  job_tab[0] = job;
  wait_job (0, status);
  return 0;
}

int put_fg_in_bg_job (void)
{
  int r;

  r = add_job (stdout, job_tab[0]);
  if (r > 0)
    job_tab[0] = NULL;
  return r;
}

int kill_job (int id, int sig)
{
  job_t l;
  int alive_procs = 0, all_stopped = 1;

  if (id < 0 || MAX_JOBS <= id || !job_tab[id])
    return -1;

  l = job_tab[id];
  while (l) {
    if (update_proc_status (l)) {
      if (kill (l->pid, sig) <0) {
        return -1;
      }
      if (update_proc_status (l))
        alive_procs = 1;
    }
    if (l->status == RUN)
      all_stopped = 0;
    l = l->next;
  }
  if (!alive_procs)
    return remove_job (id);

  if (id == 0 && all_stopped)
    return put_fg_in_bg_job ();

  return 0;
}

int kill_jobs (int sig)
{
  int id;

  for (id = 0; id < first_free; id++)
    if (job_tab[id]) {
      kill_job (id, sig);
      fprintf (stderr, "Warning: %s sent to job %d\n", signame (sig), id);
    }

  return 0;
```

```c
}

void print_proc (FILE *out, job_t l)
     /* affiche l'état du processus pointé par l
        (ne parcourt pas l!) */
{
  fprintf (out, "\t%d\t", (int) l->pid);
  switch (l->status) {
  case RUN:  fprintf (out, "Running       ");
    break;
  case STOP: fprintf (out, "Stopped");
    switch (l->code) {
    case SIGTSTP: fprintf (out, " by tty");
      break;
    case SIGSTOP: fprintf (out, "       ");
      break;
    default: fprintf (out, " by %2d ", l->code);
    }
    break;
  case EXIT:
    if (l->code)
      fprintf (out, "Exited %3d    ", l->code);
    else
      fprintf (out, "Done          ");
    break;
  case SIG:
    switch (l->code) {
    case SIGTERM: fprintf (out, "Terminated    ");
      break;
    case SIGKILL: fprintf (out, "Killed        ");
      break;
    case SIGSEGV: fprintf (out, "Seg. fault    ");
      break;
    case SIGHUP: fprintf (out, "Hanguped      ");
      break;
    case SIGPIPE: fprintf (out, "Broken pipe   ");
      break;
    default: fprintf (out, "%sed     ", signame (l->code));
    }
    break;
  default: fprintf (out, "Unknown status");
  }
  if (l->next)
    fprintf (out, "%s |\n", l->name);
  else
    fprintf (out, "%s &\n", l->name);

}

void update_print_job (FILE *out, int id)
     /* actualise et affiche les informations
        du job de n° id */
{
  job_t l;
  int alive_procs = 0;

  if (!job_tab[id])
    return;

  l = job_tab[id];
  fprintf (out, "[%d]", id);
  while (l) {
    if (update_proc_status (l))
      alive_procs = 1;
    print_proc (out, l);
    l = l->next;
  }
  if (!alive_procs)
    remove_job (id);
}

void update_print_jobs (FILE *out)
{
  int id;

  for (id = 1; id < first_free; id++)
    update_print_job (out, id);
}

void update_print_if_dead_job (FILE *out, int id)
     /* actualise et ( affiche les informations
        du job de n° id si il est vivant ) */
{
  job_t l;
  int alive_procs = 0;

  if (!job_tab[id])
    return;

  l = job_tab[id];
  while (l) {
    if (update_proc_status (l))
      alive_procs = 1;
    l = l->next;
  }

  if (!alive_procs) {
    l = job_tab[id];
    fprintf (out, "[%d]", id);
    while (l) {
      print_proc (out, l);
      l = l->next;
    }
    remove_job (id);
  }
}

void update_print_if_dead_jobs (FILE *out)
{
  int id;

  for (id = 0; id < first_free; id++)
    update_print_if_dead_job (out, id);

}
```

```c
// ---- main.c ----

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

#include "syntax.h"
#include "jobs.h"
//#include "intern.h"
#include "exec.h"


#define CWD_SIZE 64

/* gestion des signaux ^C et ^Z, mais ca marche pas ... */
/*  void handler (int sig) */
/*  { */
/*    kill_job (0, sig); */
/*  } */


int main (int argc, char **argv, char **env) {
  comm_t comm;
  int last_ret_val = 0;
  char cwd[CWD_SIZE];

/* gestion des signaux ^C et ^Z, mais ca marche pas ... */
/*    struct sigaction sa; */

/*    sa.sa_handler = &handler; */
/*    sa.sa_flags = 0; */
/*    sigemptyset (&(sa.sa_mask)); */

/*    sigaction (SIGINT, &sa, NULL); */
/*    sigaction (SIGTSTP, &sa, NULL); */

  init_job_tab ();

  printf ("Vos desirs sont des ordres, Maitre.\n");
  do {
    if (getcwd (cwd, CWD_SIZE))
      printf ("%c %s > ", (last_ret_val ? '!' : '-'), cwd);
    else
      printf ("%c /? > ", (last_ret_val ? '!' : '-'));

    fflush (stdout);
    comm = read_comm (stdin);
    update_print_if_dead_jobs (stdout);
    last_ret_val = exec_comm (comm);
    free_comm (comm);
  } while (1);

  return 0;
}
```

```c
// ---- syntax.c ----

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

#include "syntax.h"
#include "jobs.h"
#include "intern.h"


// --- LIBERATION DE L'ESPACE MEMOIRE

void free_token_list (token_list_t l)
{
  token_list_t p;
  while (l) {
    p = l;
    l = l->next;
    if (!p->used) free (p->token);
    free (p);
  }
}

void free_elt_list (elt_list_t l, free_t type)
{
  elt_list_t p;
  while (l) {
    p = l;
    l = l->next;
    if (type == HARD) free (p->elt.word);
    free (p);
  }
}

void free_scomm (scomm_t scomm, free_t type)
{
  free_elt_list (scomm.elt_list, type);
}

void free_pipe (pipe_t l, free_t type)
{
  pipe_t p;
  while (l) {
    p = l;
    l = l->next;
    free_scomm (p->scomm, type);
    free (p);
  }
}

void _free_comm (comm_t l, free_t type)
{
  comm_t p;
  while (l) {
    p = l;
    l = l->next;
    free_pipe (p->pipe, type);
    free (p);
  }
}

void free_comm (comm_t l)
{
  _free_comm (l, HARD);
}

// LECTURE D'UN FICHIER

int nb_bs_mod_2 (char *s, char *p)
     /* renvoie la parit\216 du nb de \
        qui sont avant p (inclus) dans s
     */
{
  int nb = 0;

  for (; p >= s; p--)
    if (*p == '\\')
      nb ^= 1;
    else
      return nb;
  return nb;
}

int read_line (char *s, FILE *f)
     // Lit une ligne dans f et la place dans s
     // suppos\216 de taille >= MAX_LINE_SIZE
{
  char *p;
  int len, r;

  p = s + 2;
  len = 2;


  do {
    len -= 2;
    p -= 2;
    *p = '\0';

    if (!fgets (p, MAX_LINE_SIZE - len, f)) {
      // fin de fichier ou erreur
      *s = '\0';
      return E_FGETS;
    }

    len += strlen (p);
    p = s + len;
    if (*(p - 1) != '\n') { // ligne trop longue
      if (*(p - 1) == '\\') fgetc(f);
      while ((r = fgetc(f)) != EOF &&
             r != '\n')
        if (r == '\\') fgetc(f);
      *s = '\0';
      return E_TOO_LONG;
    }
  } while (p >= s + 2 && nb_bs_mod_2 (s, p - 2));

  *(p - 1) = '\0';
  return 0;
}



char *take (char **s)
     /* prend un 'token' ("chaine", 'chaine', <, >, |, &, ;,
        ou un mot au \ pr\216s)
        dans *s et avance *s au debut du prochain token
     */
{
  char *start, *token;
```

```c
  for (; **s == ' '; (*s)++);

  start = *s;
  if (*start == '\'') {
    (*s)++;
    for (; **s && **s != '\''; (*s)++);
    if (**s != '\'') {
      fprintf (stderr, "Syntax error: ' missing.\n");
      return NULL;
    }
    (*s)++;
  } else if (*start == '"') {
    (*s)++;
    for (; **s && **s != '"' && **s != '\''; (*s)++)
      if (**s == '\\') (*s)++;
    if (**s != '"') {
      fprintf (stderr, "Syntax error: \" missing.\n");
      return NULL;
    }
    (*s)++;
  } else if (strchr ("<>|&;", *start)) {
    *s = start + 1;
  } else {
    for (; **s && (!strchr (" '\"<>|&;", **s)); (*s)++)
      if (**s == '\\') (*s)++;
  }

  token = malloc (*s - start + 1);
  strncpy (token, start, (int) (*s - start));
  token[*s - start] = '\0';

  for (; **s == ' '; (*s)++);

  return token;
}


token_list_t get_token_list (char *s)
{
  char *sp, *token;
  token_list_t l, *p;

  sp = s;
  p = &l;
  while (*sp) {
    token = take (&sp);
    if (!token) {
      *p = NULL;
      free_token_list (l);
      return NULL;
    }
    if (!strcmp (token, "")) break;
    *p = malloc (sizeof **p);
    (*p)->token = token;
    (*p)->used = 0;
    p = &((*p)->next);
  }
  *p = NULL;
  return l;
}

/* Toutes les fonctions get_xxx suivantes (sauf get_comm) renvoient le prochain
   xxx dans la token_list l (si il y en a un) et avance l sur le token suivant.
   si il n'y a pas de xxx en tete de l, elles renvoient NULL et ne touchent pas a l
 */

char *get_word (token_list_t *l)
{
  char *word;

  if (!*l) return NULL;

  word = (*l)->token;
  if (strcmp (word, "<") &&
      strcmp (word, ">") &&
      strcmp (word, "|") &&
      strcmp (word, ";") &&
      strcmp (word, "&")) {
    (*l)->used = 1;
    *l = (*l)->next;
    return word;
  }
  return NULL;
}


elt_t get_elt (token_list_t *l)
{
  token_list_t p = *l;
  elt_t elt = {ARG, NULL};

  if ((elt.word = get_word (l)))
    return elt;
  else if (!strcmp (p->token, ">")) {
    elt.type = OUT;
    *l = (*l)->next;
    elt.word = get_word (l);
  } else if (!strcmp (p->token, "<")) {
    elt.type = IN;
    *l = (*l)->next;
    elt.word = get_word (l);
  }
  if (!elt.word)
    *l = p;
  return elt;
}

elt_list_t get_elt_list (token_list_t *l)
{
  elt_list_t elt_list, *p;
  elt_t elt;

  p = &elt_list;
  while (*l) {
    elt = get_elt (l);
    if (!(elt.word)) break;
    *p = malloc (sizeof **p);
    (*p)->elt = elt;
    p = &((*p)->next);
  }
  *p = NULL;
  return elt_list;
}

scomm_t get_scomm (token_list_t *l)
{
  scomm_t scomm = {NULL, NULL};

  scomm.name = get_word (l);
  if (!scomm.name) return scomm;
  scomm.elt_list = get_elt_list (l);
```

```c
  return scomm;
}

pipe_t get_pipe (token_list_t *l)
{
  pipe_t pipe, *p;
  scomm_t scomm;

  p = &pipe;
  while (*l) {
    scomm = get_scomm (l);
    if (!(scomm.name)) break;
    *p = malloc (sizeof **p);
    (*p)->scomm = scomm;
    (*p)->pid = (pid_t) -1;
    p = &((*p)->next);
    if ((!(*l)) || (strcmp ((*l)->token, "|")))
      break;
    else
      *l = (*l)->next;
  }
  *p = NULL;
  return pipe;
}

comm_t get_comm (token_list_t l)
     /*  renvoie la commande contenue dans l,
         renvoie NULL et affiche un message
         si erreur de syntaxe
      */
{
  token_list_t tok = l;
  comm_t comm = NULL, *p;
  pipe_t pipe;

  p = &comm;
  while (tok) {
    pipe = get_pipe (&tok);
    *p = malloc (sizeof **p);
    (*p)->pipe = pipe;
    if (!tok || !strcmp(tok->token, ";"))
      (*p)->sync = 1;
    else if (!strcmp(tok->token, "&"))
      (*p)->sync = 0;
    else {
      (*p)->next = NULL;
      _free_comm (comm, SOFT);
      fprintf (stderr, "Parse error near %s\n",tok->token);
      return NULL;
    }
    if (tok)
      tok = tok->next;
    p = &((*p)->next);
  }
  *p = NULL;
  return comm;
}



// --- POUR LES TESTS


void print_token_list (token_list_t l)
{
  token_list_t p = l;

  while (p) {
    printf ("'%s' ", p->token);
    p = p->next;
  }
  printf ("\n");
}

void print_elt_list (elt_list_t l)
{
  elt_list_t p = l;

  while (p) {
    switch (p->elt.type) {
    case ARG: printf ("<");break;
    case  IN: printf ("<in: "); break;
    case OUT: printf ("<out: "); break;
    }
    printf ("%s> ", p->elt.word);
    p = p->next;
  }
  printf ("\n");
}

void print_scomm (scomm_t scomm)
{
  printf ("  - %s, ", scomm.name);
  print_elt_list (scomm.elt_list);
}

void print_pipe (pipe_t l)
{
  pipe_t p = l;

  while (p) {
    print_scomm (p->scomm);
    p = p->next;
  }
  printf ("\n");
}

void print_comm (comm_t comm)
{
  while (comm) {
    if (comm->sync)
      printf ("synchrone :\n");
    else
      printf ("asynchrone :\n");
    print_pipe (comm->pipe);
    comm = comm->next;
  }
}


// Fonction principale du module

comm_t read_comm (FILE *f)
{
  char line[MAX_LINE_SIZE];
  char *exit_argv[2] = {"0", NULL};
  int r;
  token_list_t l;
  comm_t comm;

  r = read_line (line, f);
  if (r == E_FGETS) {
```

```
    printf ("\n");
    update_print_if_dead_jobs (stdout);
    dd_exit (stdin, stdout, exit_argv);
  } else if (r == E_TOO_LONG) {
    fprintf (stderr, "Error: line too long.\n");
    return NULL;
  }
  l = get_token_list (line);
  comm = get_comm (l);
  free_token_list (l);

  return comm;
}
```

```
// ---- exec.h ----

#ifndef EXEC_H
#define EXEC_H

int exec_comm (comm_t);

#endif
```

```c
// ---- intern.h ----

#ifndef INTERN_H
#define INTERN_H

/* Ces commandes prennent en argument un flux d'entrée
   (au cas où un jour on ait envie de faire une commande
   interne faisant quelquechose avec), un flux de sortie et
   un argv au meme format que execv
*/

int dd_jobs (FILE *, FILE *, char **);

int dd_wait (FILE *, FILE *, char **);

int dd_cd   (FILE *, FILE *, char **);

int dd_exit (FILE *, FILE *, char **);

#endif
```

```
// ---- jobs.h ----

#ifndef JOBS_H
#define JOBS_H

#define MAX_JOBS 20

typedef struct st_job_cell {
  char *name;
  pid_t pid;
  enum { RUN, EXIT, STOP, SIG } status;
  int code;
  struct st_job_cell *next;
} *job_t;

void init_job_tab (void);

int can_create_new_job (void);

int exists_job (void);

job_t make_job (pipe_t);

int add_job (FILE *, job_t);

int wait_job (int, int *);

int put_in_fg_job (job_t, int *);

int put_fg_in_bg_job (void);

int kill_job (int, int);

int kill_jobs (int);

void update_print_jobs (FILE *);

void update_print_if_dead_jobs (FILE *);

#endif
```

```c
// ---- syntax.h ----

#ifndef SYNTAX_H
#define SYNTAX_H

#define MAX_LINE_SIZE 257
#define E_FGETS (-1)
#define E_TOO_LONG (-2)

typedef struct st_token_cell { //liste de token
  char *token;
  int used;
  struct st_token_cell *next;
} *token_list_t;

typedef enum {ARG, IN, OUT} elt_type; //type de l'argument : utile pour les redirections

typedef struct st_elt { //élément = mot+type
  elt_type type;
  char *word;
} elt_t;

typedef struct st_elt_cell {//liste d'éléments
  elt_t elt;
  struct st_elt_cell *next;
} *elt_list_t;

typedef struct st_scomm {
  char *name;
  elt_list_t elt_list;
} scomm_t; // une commande simple

typedef struct st_pipe {
  scomm_t scomm;
  int fd[2]; // fd : les 2 fd du tube : entrée + sortie
  pid_t pid;
  struct st_pipe *next;
} *pipe_t; // une liste de commandes simples 'pipées'

typedef struct st_comm {
  pipe_t pipe;
  int sync;
  struct st_comm *next;
} *comm_t; // une commande complete

typedef enum {SOFT, HARD} free_t; //type de libération : libération uniquement des cellules
//dans le cas de SOFT, de tout dans le cas de HARD


comm_t read_comm (FILE *);
/* lit une ligne sur le FILE* et renvoie la commande (cad liste de tubes) contenue
   dans cette ligne, si erreur de syntaxe, affiche un message et renvoie NULL
 */

void free_comm (comm_t);
/* Libère l'espace memoire occupé par la commande
 */

void print_comm (comm_t);
/* Affiche la description syntaxique de la commande
 */


#endif
```

```
CFLAGS=-g -Wall

source:= $(wildcard *.c)

include:= $(wildcard *.h)

ddsh: main.o syntax.o exec.o jobs.o intern.o
	gcc $(CFLAGS) -o ddsh main.o syntax.o exec.o jobs.o intern.o

main.o: main.c syntax.h exec.h jobs.h intern.h
	gcc $(CFLAGS) -c main.c


syntax.o: syntax.c syntax.h jobs.h intern.h
	gcc $(CFLAGS) -c syntax.c

exec.o: exec.c exec.h syntax.h jobs.h
	gcc $(CFLAGS) -c exec.c

jobs.o: jobs.c jobs.h syntax.h
	gcc $(CFLAGS) -c jobs.c

intern.o: intern.c intern.h syntax.h jobs.h
	gcc $(CFLAGS) -c intern.c

.PHONY: print clean

print:
	enscript -Ec --columns=2  --borders --media=A4 --landscape --encoding=latin1 $(sourc
e) $(include) makefile -o listing.ps
##	lpr listing.ps


clean:
	rm -vf *.o *~ \#*\# ddsh
```