THE COMMAND PROCESSING MODULE

This is, from the programming point of view, possibly the most advanced issue that we shall consider in this class. We need to create a module that will receive a string from the interface, separate it into parts (typically, the verb and the object), and execute the command it represents. Let us call it the *command processor* and, since we are computer scientists and as such we love acronyms, let us indicate it as CoP. The important point is this: *the command module (CoP) knows nothing about the game*. We want to design a generic CoP that can be used in all sorts of situation, even those that have nothing to do with the game. Now, this is at first sight a problem. Let us consider the following situation: we are giving a command such as

```
pick flashlight
```

In a game module (let us call it *game.c*, but it doesn't have to be so: it can be any other module) we have a function, let us call it *pick_fun*, which executes that command. The command module must interpret the command, separate the verb from the object, do a bunch of other stuff and *call the function pick_fun*. But... here's the rub: the function *pick_fun* is not part of the module. In the source code of CM there is no reference to *pick_fun*, nor there can be, since CM must be comletely generic and must work in the same way whatever the situation. So, if the source code of the CoP knows nothing of the function and, therefore, the object code doesn't contain the function or a link to it, there is only one possibility left: *the function must be passed to the module at runtime*.

This is where function pointers come in. We shall use them implementing a technique called *call-back*. It is a very important technique because it happens quite often that a library must execute a function that is known to it only at run-time. Think, for example, of a library that manages the GUI (graphical user interface) of a program. Your program wants to execute a certain function when the OK button is pressed; however, managing the button, the mouse, and detecting when the button is pressed is a job for the interface library, not the program. You don't have the code of the library, so you can't simply insert your function into it: you must tell the library *at run-time* that in case the OK button is pressed, you want it to call the function such-and-such. This you do using call-back. So, before taking care of the CoP, we shall have a quick look at function pointers and call-back.

FUNCTION POINTERS AND CALL-BACK

A pointer to a function is, more or less, the address of the memory location where you have to jump to execute the function. You have actually already used pointers to functions, many times: in C, the name of a function is in reality a pointer to it so whenever you call a function like *fopen* or *printf* you are actually using a pointer to a function. These pointers are fixed, and can't be changed: they are *constant* function pointers, much like the numbers *3* or *4.50* are integer or double precision constants.

Constant are boring: if we want to do something interesting with function pointers we have to declare variables and, like every variable in C, they must have a type. One could simply say:

"well... a function pointer is a variable of type *function pointer*, and that's the end of it, right?". Well, no. C is a little more specific than that: the type of a function pointer depends on the *signature* of the function it points to, that is, on the type and number of its arguments and the type of the value it returns. Types are defined with the instruction *typedef*, function types too. So, let us see how we can define some function pointer types:

```
typedef int (*myintfun)(int, int);
typedef double (*mydoublefun)(double);
```

These declarations declare two different types of function pointers: *myintfun* is a pointer to a function that accepts two integers and returns an integer, while *mydoublefun* is a pointer to a function that accepts a double precision number and returns a double precision number. Variables of these types are declared just as any other variable:

```
myintfun f1;
mydoublefun f2;
```

In order to be useful, variables need to have some value assigned to them. For a generic variable, this value can be the result of an operation (this is something very rarely done with function pointers), by assignment of another variable, or from a constant. Everything has to start with constants so... how do we create constants of these types? Well, we just declare functions. The following two functions are constants of the types we have just declared (note that it is not explicitly stated that they are values of these types: the compiler will be able to infer it comparing the function declaration with the type definition):

```
int sum(int i, int j) {
    return i+j;
}

double db(double x) {
    return 2*x;
}
```

Not the most exciting functions, but they will do. Now we can assign their value to pointers:

```
f1 = sum;
f2 = db;
```

How do we use function pointers? Well, we use them by executing the function that they point to, and this we can do by extracting the value of the pointer (using the operator "*", as with any pointer), and passing it parameters:

```
int q = (*f1)(2, 3);
double p = (*f2)(3.0);
```

Note that there is a small syntactic inconsistency here: the names of a functions are constant function pointers so, according to the syntax of function pointers, a function like *sum* should be used as

```
int t = (*sum)(2, 3);
```

This would be quit einconvenient, so in C we use the (incorrect but more practical) alternative syntax[1].

So far we haven't really gotten anywhere: we can call a function using a pointer to it stored in a variable but, so what? Things start getting interesting if we consider that variables (even function pointers) can be passed to functions. So, I can think of a generic execution function like

```
int funexe(myintfun f, int a, int b) {
     return (*f)(a, b);
}
```

So now we can call our function passing it as a parameter the pointer to the function we want to compute. See the code in figure 1. Executing this program with the parameters 3 and 2 will print *5, 6* (the sum of 2 and 3 and their product).

```
> test 2 3
5 6
>
```

This is still not terribly interesting, but how about this: now on the command line we want to give three numbers: the first two are the arguments of the function as before, the third will indicate which one of the functions will be computed. This can easily be done by introducing an array of function pointers as in figure 2.

With this modification we can have, for example:

```
> test 10 8 1
18
> test 10 8 2
80
> test 10 8 3
2
>
```

We can apply the same principle in order to associate to each function a *name* rather than an index. In this case, it is convenient to create a structure (let us call it *funassoc*) that contains a name and the associated function pointer. We create an array of such structures and we have the function funexe search it for an entry whose name matches the name we have given. That is the entry corresponding to the name we want, now all we have to do is to call the corresponding function pointer. The idea is implemented in figure 3

With this we can do

```
> test 10 8 add
18
> test 10 8 multiply
80
> test 10 8 modulo
2
>
```

[1] I am not quite sure whether the correct syntax would work: I confess that I never tried it. It could work.

```
typedef int (*myintfun)(int, int);

int sum(int a, int b) {
  return a+b;
}
int mult(int a, int b) {
  return a*b;
}
int mmod(int a, int b) {
  return a % b;
}

int funexe(myintfun f, int a, int b) {
  return (*f)(a, b);
}

int main(int argc, char **argv) {
  myintfun f1, f2;
  int res1, res2;
  int a, b;

  a = atoi(argv[1]);
  b = atoi(argv[2]);
  f1 = sum;
  f2 = mult;
  res1 = funexe(f1, a, b);
  res2 = funexe(f2, a, b);
  printf("%d %d\n", res1, res2);
}
```

Figure 1: Passing function pointers to a function

The key here is that the function *funexe*, which is the one that calls the functions, can be completely oblivious of what functions are defined until run-time. All it has to know is the *type* of the function pointer to be used. If we want to put the function *funexe* in a separate file and compile separately, all we have to know at compile time is the type *myintfun*, not the specific functions that will have to be computed: the function *funexe* is completely general, I can use it to execute any kind of function (as long as their signatures are the same). If (as is the case of CoP) deciding what kind of function to call and with what values of the parameters depends on a complex logic (in our case this includes interpreting the command line as given by the player), I can implement this logic in the funciton *funexe*, while I define the consequences of the actions (viz. the functions that are executed as a consequence of the decision) outside of it. This is the principle that we shall use in order to design a general-purpose command execution module.

```
typedef int (*myintfun)(int, int);

int sum(int a, int b) {
  return a+b;
}
int mult(int a, int b) {
return a*b;
}
int mmod(int a, int b) {
  return a % b;
}

int funexe(myintfun f, int a, int b) {
  return (*f)(a, b);
}

int main(int argc, char **argv) {
  myintfun *f;
  int res;
  int idx;
  int a, b;

  a = atoi(argv[1]);
  b = atoi(argv[2]);
  idx = atoi(argv[3]);
  f = (myintfun *) malloc(3*sizeof(myintfun));
  f[0] = sum;
  f[1] = mult;
  f[2] = mmod;
  res = funexe(f[idx], a, b);
  printf("%d\n", res);
}
```

Figure 2: Passing function pointers to a function with run-time function selection

```
typedef int (*myintfun)(int, int);

typedef struct {
  char *name;
  myintfun f;
} funassoc;

int sum(int a, int b) { return a+b; }
int mult(int a, int b) { return a*b; }
int mmod(int a, int b) { return a % b; }

int funexe(funassoc **flist, char *fname, int n, int a, int b) {
  for (int i=0; i<n; i++) {
    if (!strcmp(flist[i]->name, fname)) {
      return}(*(flist[i]->f))(a, b);
  }
  printf("Error:  unknown function %s\n", fname);
  exit(1);
  return 0;
}

int main(int argc, char **argv) {
  funassoc **flist;
  int res;
  char *fname;
  int a, b;
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  fname = argv[3];
  flist = (funassoc **) malloc(3*sizeof(funassoc *));
  flist[0] = (funassoc *) malloc(sizeof(funassoc));
  flist[0]->name = strdup("add");
  flist[0]->f = sum;
  flist[1] = (funassoc *) malloc(sizeof(funassoc));
  flist[1]->name = strdup("multiply");
  flist[1]->f = mult;
  flist[2] = (funassoc *) malloc(sizeof(funassoc));
  flist[2]->name = strdup("modulo");
  flist[2]->f = mmod;
  res = funexe(flist, fname, 3, a, b);
  printf("%d\n", res);
}
```

Figure 3: Passing function pointers to a function with run-time function selection using a function name

THE CoP MODULE FROM OUTSIDE

So, how are we going to use these ideas to build a commend module? Ultimately we want to associate a command that the user types (something like "turn right") into a couple of things: a call to a suitable function (let us say the function *turn_implementation*) with the right parameter (which, in this case, will be the string "right". (We shall see soon that we shall need to add more parameters, but for the moment this one will suffice). So, the user will type something in the form

<verb> <predicate>

and we want to use the verb to call a function (much like we have done in the last example above), and the predicate into an argument for the function.

We shall use, in this case, a more indirect method than we have done above. Instead of associating directly the name "turn" to a function pointer (in a structure like the *funassoc* that we have used in figure 3), we shall associate the function pointer with an *internal name*, say "loc_dir_change" (I am just making names up). Our association will then require us to consult two tables: first we will have to see that the command "turn" corresponds to the internal name "loc_dir_change" using one table; then, using the second table, we shall associate the local name "loc_dir_change" to the appropriate function pointer. The reason for this roundabout way of doing things will be clear, I hope, in the following.

In the last example above, the main program would allocate and manage the array of structures that associated the function names with the corresponding function pointers. In the CoP, we shall relieve the calling program from this burden, and have the CoP itself take care of this management. The calling program will simply call a function that will create, inside the CoP, an association between a name and a function. Let assume that we have a CoP data structure defined somewhere:

```
typedef struct {
   .
   .  /* something here... we'll see */
   .
} CoP
```

and a program that creates a command processor:

```
CoP *c = CoP_Create(....)  /* We'll see the parameters */
```

Now we define a function to execute, say, the *go* command:

```
int go_execute(....) {  /* we'll see the parameters later */
  .
  .
}
```

If we want to associate this function to an internal name, say "go_cmd_int" we do a call like:

```
int status = CoP_associate(CoP *c, char *name, CoP_function f);
```

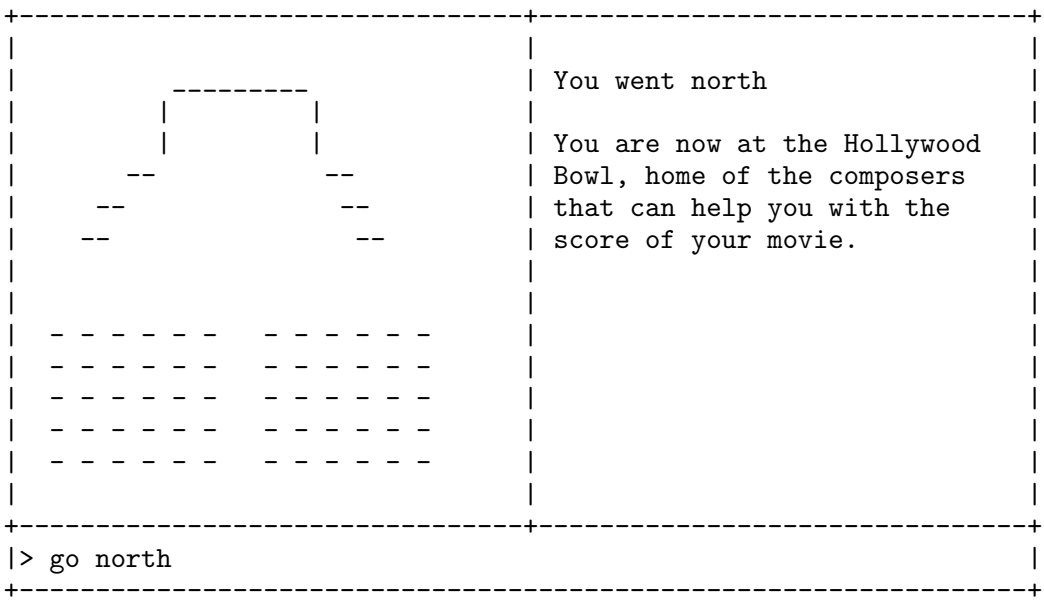
where Cop_function is a function pointer data type, a pointer to the functions that execute commands.

Let us now consider the command functions a bit up-close and personal. We need to do this now because we shall have to define the pointer to them. What parameter sdo they accept? Let us consider the function that execute a command like

go north

The verb *go* will be analyzed by the CoP to determine which function must be called. This function, in order to execute the command, will have to know where the player is in order to determine whether it can move north and, in case it can where it will move to. In order to do this, the function will have to receive a pointer to the word structure, which contains the whole information about the game. The function will also have to receive the predicate of the command, that is, the word *north*, to know in which direction to move. This is true for all functions, of course: a function that executes the command *pick*, for example, will have to receive the name of the thing that it has to pick, and the pointer to the world to know whetehr the thing to pick is in the same space as the player and to move it from the list of object in the space to the list of things in possession of the player.

So we already have two parameters: a pointer to the world, and a string that contains the object of the command. We have to consider another point, though: it is possible that the command will have to write something on the screen depending on how the execution goes. We don't want these things to be hard-coded into the program: we want the program to read them from a file so that we can easily change them without recompiling the program (and, mor importantly, we find them all in the same place and we don't have to chase them all over the code). For example, if we have been successful in going north, the function *go_execute* might want to write om the interface something like "you have moved north", possibly followed by a description of the environment in which we are now. Something like:



The description of the environment has been read from a different file, the file from which we read all the environments, but how do we get the first sentence, the one that says that we go north? Note that that sentence is not unique, it depends on the object that we have given in

the command: had we said *go south*, the sentence would have read "You went south". Also, if for some reason we couldn't go north, the function would have to print something like "You can't go north!"

There are several options (one of them is to create special files, one for each command, with the sentences that the command might need to use); here we shall consider a simple one: the sentences come from the same file where we read the name of the command and its association to the internal name.

Let us say that we are defining the command *pick*, and that the command has three possible answers: you either picked an object (and the command prints a confirmation), or you can't pick the object because the object is not there, or, finally, you can't pick the object because you need more strength in order to pick it. We want to put these three sentences in the command file, right when we define the command. Since we don't know what the object will be (it depends on what the object of the command is), we put a placeholder in its stead. SO, in the file that contains all commands, the definition of *pick* will be something like this:

```
pick
pick_cmd_int
3 # this is the name of sentences that we are going to write
You have picked up the *
There is no * here!
You can't pick up the *: you need more strength
```

The first line is the name of the command, the thing that we shall write on the screen; then we have the internal nale, the number of possible sentences that we are going to write and finally the sentences with the placeholder where the command name is going to be.

These sentences will be passed to the command execution functions. Each one of them will therefore have the following type:

```
typedef int (*cmdfun_type)(void *, char *, cahr **, int n);
```

What happens if you give the name of a command taht doesn't exist? Well, the CoP must write something on the screen using the interface. If it receives the command

```
fubar light
```

The CoP will write something like

```
I don't know how to fubar!
```

But the CoP doesn't know how to access the interface! The CoP doesn't know anything except calling commands. Well, the solution is to create a standard command, let us call it *error* that is never executed from the command line, but that is executed every time the CoP finds a verb it doesn't recognize. This command can be defined in the command file just like the other ones:

```
error
error_cmd_int
1
I don't know hot to *
```

Note that in the case of normal commands, the CoP replaces the * in the string(s) with the object that was given by the player, while in this case it will replace it with the unknown verb.

* * *

How would the CoP work from outside? We need three functions: the first one creates the CoP, reads the contents of the file that contains the commands, and returns a pointer to the CoP:

```
CoP *CoP_create(FILE *cmdfile);
```

The second is used to create an association: the calling program will call it as many times as necessary to associate all the command functions it defines with internal command names (which correspond to the internal command names given in the commands file (one of these associations must be the function that prints out the error messages):

```
int CoP_assoc(CoP *c, char *int_name, cmdfun_type cfun);
```

(I assume that the function returns the number of associations done so far; so, the first time you call it it will return 1, the econd time it will return 2, and so on... not great information, but might be useful in some cases; the funciton returns 0 if something went wrong).

Finally, there will be a function that executes commands. This will receive the command string, as the player typed it, a pointer to the world (cast to void, since the CoP knows nothing about worlds or anything: the function will simply receive the pointer and pass it along to the appropriate command execution without doing anything with it), and the pointer to the CoP:

```
int CoP_execute(CoP *c, char *cmd, void *pt);
```

A very simple program that uses the CoP is shown in figure 5. The program defines three commands (one of them is the error) and a CoP that executes them. Note that we define a structure *world* that does absolutely nothing: nobody uses it: it is not even allocated. This structure has been defined to highlight an important point. The CoP knows nothing about worlds, spaces, and players, so it receives a pointer to a void and passes it along to the execution functions. This is because the CoP is compiled as a separate module, independently of the game (imagine that you have commissioned your CoP to a foreign company, which knows nothing about your game). However, the command execution functions are define inside the game module, and they *do* know about worlds and players. So, when they receive the void pointer, they can cast it to a world pointer and use it to access the world.

The difference between the CoP and the game module is syntactic: they both have access to a pointer that contains the address where the world is stored, but the CoP, not habing access to the structure declaration, can't access anything in that memory area, while the execution functions, which do have access to the declaration, can.

This program is useless unless we give it a command file to work with. There is one in figure ?? (I assume that you used my file reading functions, so I took the liberty to add comments). With this, you can have a meaningful interaction with your program, like:

```
#include <CoP.h>

typedef struct {
     int n;
} world;

int cmd1(void *dummy, char *obj, char **str, int n) {
     world *q = (world *) dummy;
     printf("cmd1:  %s\n", str[0]);
}
int cmd2(void *dummy, char *obj, char **str, int n) {
     printf("cmd2:  %s\n", str[0]);
}
int err(void *dummy, char *obj, char **str, int n) {
     printf("error:  %s\n", str[0]);
}

main() {
     char buf[100];
     world *w;
     FILE *fp = fopen("cmdfile.txt", "r");
     CoP *c = CoP_create(fp);
     CoP_assoc(c, "cmd1_internal", cmd1);
     CoP_assoc(c, "cmd2_internal", cmd2);
     CoP_assoc(c, "error_internal", err);

     while(1) {
       printf("> ");
       fflush(stdout);
       fgets(buf, 100, stdin);
       CoP_execute(c, buf, (void *) w);
     }
}
```

Figure 4: A simple execution of the CoP.

```
4        # Number of commands placed in this file...  makes parsing easier.
# First command:
dance          # This is the command that you must give in the terminal
cmd1_internal   # The internal name to which it is associates
1
Yes, let's dance a *     # The answer that it will give you

# Second command:
sing            # This is the command that you must give in the terminal
cmd1_internal     # Note:  it execute the same function!
1
Yes, let's sing * together     # But it writes a different thing

# Second command:
play
cmd2_internal       # This does execute the other function
1
I would LOVE to play *

# And finally the error:
error             # This is a standard name recognized by the CoP: it has to be this
error_internal      # This is the error function that you have declared
1
Sorry...  I don't know what * is
```

Figure 5: A simple command file.

```
> dance Tango
Yes, let's dance a Tango
> sing a lullaby
Yes, let's sing a lullaby together
> play guitar
I would LOVE to play guitar
> smurf the world
Sorry... I don't know what smurf is
```

Neat, ain't it?

The problem, of course, is how to do it, that is, how to program the CoP. It is actually quite easy. Here are some indications.

UNDER THE HOOD

So, let's have a look at how things work. The CoP structure needs quite a few things. Mainly it will contain two lists of structures: the first list will contain the information that we have collected from the command file. Each structure will contain the information about a command, basically the name of the command, its internal name, and all the strings that it can print. It will be something like this:

```
typedef struct {
   char *cmd;
   char *internal;
   int  n_msg;
   char **msg;
} ext_cmd;
```

When we create the CoP, the create function will read the command file and allocate an array of pointers to these stuctures, one for each command that we have defined. One project issue: will the error function be an entry in this array, or will it have its separate pointer inside the CoP structure? There are good arguments to be made in favor and against both these options. Here I will assume that the structure that defines the error function has a separate pointer. So when I read that in a file there are n commands defined, I will only allocate space for n-1 pointers in the array: one of them will be the error function and will have its own pointer.

A second structure that will be necessary associates the internal name of a function to the corresponding function pointer. This structure is simpler:

```
typedef struct {
    char *i_name;
    cmdfun_type fct;
} int_cmd;
```

The CoP structure also has a list of pointers to these functions but, at the begining, this list will be empty, as there are no associations yet: this list will be populated, one element at the time, as the program calls the function *CoP_assoc*. The complete CoP structure might look something like this:

```
typedef struct {
```

```
    ext_cmd **e_lst;  /* The list with the "normal" commands... */
    int     ext_no;   /* ...and the number of elements in it    */
    ext_cmd *error;   /* The "special" error command            */
    int_cmd **i_lst;  /* List with associations between internal commands and functions */
    int     int_no;   /* and the number of elements it contains (0 at the beginning) */
    int     int_max;  /* The number of pointers i_lst that are actually allocated    */
} CoP;
```

The *create* function will create the structure, read the command file, and create a list of *ext_cmd* strctures, one for each command specified in the file. It might also allocate the initial *i_lst* list: in my design, I do an initial allocation of the list with a certain number of pointers (say..10) and store this number in *int_max*. The number of elements actually used (*int_no*) is set to 0. As I call the *assoc* function, I create internal structures, use the pointers, and incerment *int_no*. When I run out of space U do a *realloc* on the list (see the function below).

The *CoP_assoc* function will create a new association:

```
/*
 Local function: searches for an association given an internal name;
 returns a pointer to the associated function, or NULL if the
 internal name is not found.
*/
static cmdfun_type _assoc_search(CoP *c, char *i_name) {
     for (int i=0; i<c->int_no; i++) {
         if (!strcmp(i_name, c->i_lst[i]->i_name)) {
             return c->i_lst[i]->fct;
         }
     }
     return NULL;
}


int CoP_assoc(CoP *c, char *int_name, cmdfun_type cfun) {
      if (_assoc_search(c, int_name) != NULL) {
          return -1;                  /* the internal command already existed */
      }
      if (c->int_no >= c->int_max-1) {  /* list full: get more space   */
          c->int_max += 10;
          c->i_lst = (int_cmd **) realloc(c->i_lst, c->int_max);
      }
      int_cmd *new_a = (int_cmd *) malloc(sizeof(int_cmd));
      new_a->i_name = strdup(int_name);
      new_a->fct = cfun;
      c->i_lst[c->int_no++] = new_a;
      return c->int_no;
}
```

The function that executes the commands is also very simple, and uses the same internal function *_assoc_search* that we have already defined:

```
static ext_cmd *_ext_src(CoP *c, char *name) {
   /* Ok... this is easy... I won't implemet it */
```

```
}

/*
    Prepares the strings of an external command for execution: returns
    an array of prepared strings with the starts removed and replaced
    by the name of the predicate
*/
static char **_unpack_all(ext_cmd *e, char *obj) {
     char **str = (char **) malloc(e->n_msg*sizeof(char *));
     for (int i=0; i<e->n_msg; i++)
          str[i] = _unpack(e->msg[i], obj);
     return str;
}



int CoP_execute(CoP *c, char *cmd, void *pt) {
    /*
       Break the command in a verb and a predicate.
       Easy... I won't do it
    */
    char *verb;
    char *obj;


    /* First, search the list of external commands for the right one */
    ext_cmd *e = _ext_src(c, name);
    if (!e) {
       char **str = _unpack_all(c->error, verb);
       cmdfun_type f = _assoc_search(c, c->error->internal);
       if(!f) {
         abort("catastrophic error");
       }
       return (*f)(pt, verb, str, c->error->n_msg);
       /* There are some frees to do before returning... */
    }
    /* The command has been found */
    char **str = _unpack_all(e, obj);
    cmdfun_type f = _assoc_search(c, e->internal);
    if(!f) {
      abort("catastrophic error");
    }
    return (*f)(pt, obj, str, e->n_msg);
    /* There are some frees to do before returning... */
}
```

There are a few parts missing that you will have to fill in. The most relevant is reading the commands file and creating the list of external commands. The rest is pretty easy. Well.. the function *_unpack* may be a trifle tricky, as strings may contain several stars... OK... here it is:

```
char *_unpack(char *str, char *repl) {
   char *ret, *pt;
   int n = 0;
```

```
   /* count the number of stars in the first string */
   for (p=str; *p; p++) {
     if (*p == '*') n++
   }

   /* allocates the array with enough space for the final string (I add
      1 to store the final zero) */
   ret = (char *) malloc(str + n*repl);
   p = ret;
   while (*str) {
     if (*str!='*') {
       *p++ = *str++;
     }
     else {
       strcpy(p, repl);
       p += strlen(repl);
       str++;
     }
   }
   return ret;
}
```

Well... that is pretty much all... **Good Work!**

And, of course, don't hesitate to come and see me if you have any doubts. This is a fairly tricky piece of code so it is **very** important that you clarify all doubts that you might have. Come and stop by. I won't bite you.