CE437: Αλγόριθμοι CAD I

Homework 2

Tcl shell's Implementation

By Vasileiou Christos, 1983

Files' Structure

- customTCL.c: Includes main implemenation.
    - int main(int argc, char *argv[])
- Instructions.h: Includes Tcl instructions in a string array.
    - static char *instructions[]
- functions_1st.c: Includes 1st homework's functions.
- functions_2nd.c: Includes 2nd homework's functions.
    - void *commandsCreation();
    - Tcl_ObjCmdProc *cube_intersect_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[])
    - Tcl_ObjCmdProc *supercube_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *distance_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *cube_cover_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *sharp_2 (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *my_sharp (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - void *sharp (ClientData clientData, Tcl_Interp *interp, int objc, Tcl_Obj *CONST objv[]);
    - int checkIfValid (char *checked, int size);
- Makefile: Linking and Compilation.

commandsCreation() function

```
// Create 5 new cubes' commands //
void *commandsCreation()
{
    /*********************************************************************
     * The TclStubs mechanism defines a way to dynamically bind          *
     * extensions to a particular Tcl implementation at run time.        *
     * This provides two significant benefits to Tcl users:              *
     *  1. Extensions that use the stubs mechanism can be loaded into    *
     *     multiple versions of Tcl without being recompiled or relinked. *
     *  2. Extensions that use the stubs mechanism can be dynamically    *
     *     loaded into statically-linked Tcl applications.               *
     *********************************************************************/

    if (Tcl_InitStubs(tcl_interpreter, TCL_VERSION, 0) == NULL)
    {
        return NULL;
    }
    Tcl_CreateObjCommand ( tcl_interpreter, "cube_intersect_2", ( Tcl_ObjCmdProc* ) cube_intersect_2, NULL, NULL );
    Tcl_CreateObjCommand ( tcl_interpreter, "supercube_2", ( Tcl_ObjCmdProc* ) supercube_2, NULL, NULL );
    Tcl_CreateObjCommand ( tcl_interpreter, "distance_2", ( Tcl_ObjCmdProc* ) distance_2, NULL, NULL );
    Tcl_CreateObjCommand ( tcl_interpreter, "cube_cover_2", ( Tcl_ObjCmdProc* ) cube_cover_2, NULL, NULL );
    Tcl_CreateObjCommand ( tcl_interpreter, "sharp_2", ( Tcl_ObjCmdProc* ) sharp_2, NULL, NULL );
    Tcl_CreateObjCommand ( tcl_interpreter, "sharp", ( Tcl_ObjCmdProc* ) sharp, NULL, NULL );

}
```

checkIfValid implementation

```
int checkIfValid ( char *checked, int size )
{
    int j;

    NotValid = 0;
    for ( j = 0; j < size; j = j+2 )
    {
        if ( (strncmp ( &checked[j], "00", 2) == 0) )
        {
            // found not valid cubes "00" //
            NotValid++;
        }
    }

    if ( NotValid > 0)
        return NotValid; // return the number of not valid's cubes. //

    return NotValid;
}
```

checkIfValid:
returns the number of invalid patterns.

Reading arguments

```
if (objc != 3)
{
    fprintf( stderr, "supercube's usage needs exactly 2 arguments\n" );
    return NULL;
}

int j, size, validity;
size = strlen ( Tcl_GetString ( objv [1] ) );  // argument's length //
if (size%2 != 0)
{
    fprintf( stderr, "supercube's arguments must have even size\n" );
    return NULL;
}

cubeSuper_2 = (char*) malloc ( size+1);
initString ( cubeSuper_2, size+1 );
char arguments [objc-1][size+1];

for ( j = 1; j < objc; j++)
{
    // all arguments have the same size as first //
    if ( strlen ( Tcl_GetString( objv [j]) ) != size )
    {
        fprintf( stderr, "supercube's arguments must have the same size\n" );
        return NULL;
    }
    initString ( arguments [j-1], size+1 );
    strncpy( arguments [j-1], Tcl_GetString ( objv[j] ), size );
    strncpy( &arguments [j-1][size], "\0", 1 );
    printf("Argument %d: %s\n", j, arguments [j-1] );
}
```

```
validity = checkIfValid ( arguments [0], size );
if ( validity > 0 )
{
    if ( sharpIsActive == 0 && sharp_2IsActive == 0)
    {
        printf ("Argument %s is not valid\n", arguments [0]);
        return NULL;
    }
}

validity = checkIfValid ( arguments [1], size );
if ( validity > 0 )
{
    if ( sharpIsActive == 0 && sharp_2IsActive == 0)
    {
        printf ("Argument %s is not valid\n", arguments [1]);
        return NULL;
    }
}
```

cube_intersect_2 and supercube_2 implementation

```
// Code below will be executed, only if sharp_2 or sharp is active. //
if ( sharp_2IsActive == 1 || sharpIsActive == 1)
{
    // 2nd argument for sharp, inverting its bits. //
    for ( j = 0; j < size; j++ )
    {
        if ( strncmp ( &arguments[1][j], "1", 1 ) == 0 )
        {
            strncpy ( &arguments[1][j], "0", 1);
        }
        else if ( strncmp ( &arguments[1][j], "0", 1 ) == 0 )
        {
            strncpy ( &arguments[1][j], "1", 1);
        }
    }
    strncpy ( &arguments[1][j], "\0", 1);
}
for ( j = 0; j < size; j++ )
{
    // cubeIntersect_2 has the value of LOGIC AND between arguments' bits //
    if ( (strncmp ( &arguments [0][j], "1", 1) == 0) && (strncmp ( &arguments [1][j], "1", 1) == 0) )
        strncpy ( &cubeIntersect_2 [j], "1", 1);
    else
        strncpy ( &cubeIntersect_2 [j], "0", 1);
}

strncpy ( &cubeIntersect_2[j], "\0", 1 );
validity = checkIfValid ( cubeIntersect_2, size );
if ( validity > 0 )
{
    printf ("Intersect %s is not valid\n", cubeIntersect_2 );
    return NULL;
}
printf ("Cube's intersect: "GREEN"%s"WHITE " \n", cubeIntersect_2);
```

```
for ( j = 0; j < size; j++ )
{
    // cubeSuper_2 has the value of LOGIC OR between arguments' bits //
    if ( (strncmp ( &arguments [0][j], "1", 1) == 0) || (strncmp ( &arguments [1][j], "1", 1) == 0)
        strncpy ( &cubeSuper_2 [j], "1", 1);
    else
        strncpy ( &cubeSuper_2 [j], "0", 1);
}
```

distance_2 and cube_cover_2 implementation

- distance_2:

```
if (objc != 3)
{
    fprintf( stderr, "distance's usage needs exactly 2 arguments\n" );
    return NULL;
}
cube_intersect_2( clientData, interp, objc, objv );

printf("Cubes' Distance is: %d\n", NotValid);
return NULL;
```

- cube_cover_2:

```
for ( j = 0; j < size; j++ )
{
    // cubeCover_2 has the value of LOGIC OR between arguments' bits //
    if ( (strncmp ( &arguments [0][j], &arguments [1][j], 1) >= 0) ) // for each bit argument0 >= argument1
        continue;
        // strcpy ( &cubeCover_2 [j], &arguments [0][j] );
    else
    {
        if ( sharpIsActive == 1)
            printf ( "%s doesn't cover %s\n", arguments [0], arguments [1] );
        return NULL;
    }
}

strcpy ( cubeCover_2, arguments [0] );
printf ( "%s covers %s\n", cubeCover_2, arguments [1] );

free (cubeCover 2);
```

Sharp_2 implementation

- sharp_2:

```
if (sharp_2IsActive == 1)
{
    printf ("sharp_2 is activated\n");
}
my_sharp ( clientData, interp, objc, objv);
```

- my_sharp:

```
char cubeSharp_2 [size][size+1];
cube_intersect_2 ( clientData, interp, objc, objv );

for ( i = 0; i < size; i++)
{
    initString ( cubeSharp_2[i], size+1 );
    for ( j = 0; j < size; j++ )
    {
        if ( i == j )
        {
            // cubeSharp_2's diagonal has the intersect's value  //
            strcpy ( &cubeSharp_2 [i][j], &cubeIntersect_2 [j] );
        }
        else
        {
            // cubeSharp_2's other cells have the 1st argument's value  //
            strcpy ( &cubeSharp_2 [i][j], &arguments [0][j]) ;
        }
    }
    strncpy ( &cubeSharp_2 [i][j], "\0", 1);
}
```

```
printf ("Cube's sharp: \n");
validity = 0;
countSharpRows = 0;
for ( i = 0; i < size; i++)
{
    // for each row checks whether is valid or not. //
    strncpy ( &cubeSharp_2 [i][size], "\0", 1);
    if ( sharpIsActive == 1 )
    {
        validity = checkIfValid ( cubeSharp_2 [i], size );
        if ( validity == 0)
        {
            cubeSharp [countSharpArrays][countSharpRows] = (char*) calloc ( size+1, sizeof(char) );
            if ( cubeSharp [countSharpArrays][countSharpRows] == NULL )
            {
                fprintf (stderr, "Calloc error\n");
            }
            initString ( cubeSharp [countSharpArrays][countSharpRows], size+1 );
            strncpy ( cubeSharp [countSharpArrays][countSharpRows], cubeSharp_2 [i], size );
            strncpy ( &cubeSharp [countSharpArrays][countSharpRows][size], "\0", 1 );
            printf (GREEN"%s"WHITE" \n", cubeSharp [countSharpArrays][countSharpRows] );
            countSharpRows++;
        }
    }
    else
    {
        printf (GREEN"%s"WHITE " \n", cubeSharp_2 [i]);
    }
```

Sharp implementation: storing arguments from list

```c
// objv[1] is 1st cube and objv[2] ... objv[objc-1] are cubes' list. //
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * argumentSharp is a string and it has in store all arguments objv[]*
 * argumentSharp = "string1\0string2\0string3\0"                      *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
char *list;
char *searchWhiteSpace, *prevWhiteSpace;
Tcl_Obj **objvArgs;

argumentSharp = (char*) realloc (argumentSharp, (size+1) * sizeof(char) );
initString ( argumentSharp, size+1 );
strncpy ( argumentSharp, Tcl_GetString ( objv[1] ), size );
strncpy ( ( argumentSharp + size ), "\0", 1 );
printf ("%s\n", ( argumentSharp ) );

sizeArg2 = strlen ( Tcl_GetString ( objv[2] ) );
list = (char*) malloc ( sizeArg2 + 1 );
strncpy ( list, Tcl_GetString ( objv[2] ), sizeArg2 );
strncpy ( &list[sizeArg2], "\0", 1 );
prevWhiteSpace = list;
// found all argumentSharp in the list { ... } //
listArgs = 0;
```

```c
for ( searchWhiteSpace = list; ( searchWhiteSpace - list ) < strlen(list); searchWhiteSpace++)
{
    // when searchWhiteSpace doesn't point anymore to digit, store it on argumentSharp. //
    if ( isdigit (*searchWhiteSpace) == 0 )
    {
        // sharp's arguments have the same size as first //
        listArgs++;
        argumentSharp = (char*) realloc ( argumentSharp, (listArgs+1) * (size+1) * sizeof(char) );
        strncpy ( ( argumentSharp + listArgs*(size+1) ), prevWhiteSpace, size);
        // Put terminating character. //
        strncpy ( ( argumentSharp + listArgs*(size+1) + size), "\0", 1 );
        prevWhiteSpace = searchWhiteSpace+1;
        printf ("%s\n", (argumentSharp+listArgs*(size+1) ) );
    }
}
if ( isdigit (*searchWhiteSpace) == 0 )
{
    listArgs++;
    argumentSharp = (char*) realloc ( argumentSharp, (listArgs+1) * (size+1) * sizeof(char) );
    strncpy ( ( argumentSharp + listArgs*(size+1) ), prevWhiteSpace, size);
    // Put terminating character. //
    strncpy ( ( argumentSharp + listArgs*(size+1) + size), "\0", 1 );
    prevWhiteSpace = searchWhiteSpace+1;
    printf ("%s\n", (argumentSharp + listArgs * (size+1) ) );
}
```

Sharp implementation: sharp_2 is activated.

```
// 3 is the arguments' number is going to pass in cube_intersect_2. //
objvArgs = (Tcl_Obj**) malloc ( 3 * sizeof(Tcl_Obj*) );
objvArgs[0] = objv[0];
objvArgs[1] = Tcl_NewStringObj ( argumentSharp, size );
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * listArgs is the arguments' number in the list {...}.  *
 * cubeSharp points to listArgs different arrays.        *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
cubeSharp = (char***) calloc ( listArgs, sizeof(char**) );
if ( cubeSharp == NULL )
{
    fprintf (stderr, "Calloc error\n");
}
countSharpArrays = 0;
// distributive property. //
for ( i = 1; i <= listArgs; i++)
{
    printf ("\n%d. Iteration\n", i);
    objvArgs [2] = Tcl_NewStringObj ( ( argumentSharp + i * (size+1) ), size );
    cubeSharp [countSharpArrays] = (char**) calloc ( size, sizeof(char*) );
    if ( cubeSharp [countSharpArrays] == NULL )
    {
        fprintf (stderr, "Calloc error\n");
    }
    my_sharp ( clientData, interp, 3, objvArgs );
    countSharpArrays++;
}
```

Sharp implementation: cube’s intersection

```c
// array has all intersection's values from distributive property. //
// Its size will be at most size ^ listArgs. //
array = (char**) calloc ( pow (size, listArgs), sizeof(char*) );
cnt = 0;
// Deactivate sharpIsActive's flag because it is responsible for inverting the second argument. //
sharpIsActive = 0;
sizeArray = 0;
// i runs for each sharp array. //
for ( i = 0; i < countSharpArrays-1; i++ )
{
    /* k runs for each row of first array that we want to calculate *
     * the intersect with each row of second array.                 */
    if (i == 0)
    {
        sizeArray = size;
    }
    else
    {
        sizeArray = sizeArray*(cnt+1);
    }
    for ( k = 0; (k < sizeArray) && (cubeSharp [i][k] != NULL); k++ )
    {
        if ( strncmp ( &cubeSharp [i][k][0], "\0", 1 ) == 0 )
        {
            continue;
        }
        // At first time get string from the first array. //
        if ( i == 0)
        {
            objvArgs [1] = Tcl_NewStringObj ( cubeSharp [i][k], size );
        }
        else // Get string by already existing array. (array has intersect's value) //
        {
            objvArgs [1] = Tcl_NewStringObj ( array [k], size );
        }
```

```c
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
         * j runs for each row of second array if there is not cube              *
         * (first cell will be '\0') or if there is not allocated block (NULL).  *
         * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
        for ( j = 0; (j < size) && (cubeSharp [i+1][j] != NULL); j++ )
        {
            if ( strncmp ( &cubeSharp [i+1][j][0], "\0", 1 ) == 0 )
            {
                continue;
            }
            objvArgs [2] = Tcl_NewStringObj ( cubeSharp [i+1][j], size );
            cube_intersect_2 ( clientData, interp, 3, objvArgs );
            validity = checkIfValid ( cubeIntersect_2, size+1);
            if ( validity > 0)
            {
                printf (RED"%s"WHITE" \n", cubeIntersect_2);
                continue;
            }
            array [cnt] = (char*) calloc ( size+1, sizeof(char) );
            if ( array [cnt] == NULL )
            {
                fprintf (stderr, "Calloc error\n");
            }
            initString ( array [cnt], size+1 );
            strncpy ( array [cnt], cubeIntersect_2, size);
            strncpy ( &array [cnt][size], "\0", 1);
            // printf ("i: %d, k: %d, j: %d\n", i, k, j);
            cnt++;
        }
        sizeArray = cnt;
    }
}
```

Sharp implementation: cube's covering

```
for ( i = 0; i < cnt; i++ )
{
    for ( j= 0; j < cnt; j++ )
    {
        if ( (i == j) || (array[i][0] == '\0') || (array[j][0] == '\0') )
        {
            continue;
        }
        objvArgs [1] = Tcl_NewStringObj ( array [i], size );
        objvArgs [2] = Tcl_NewStringObj ( array [j], size );
        cube_cover_2( clientData, interp, 3, objvArgs );
        if ( cover == 1 )
        {
            initString ( array [j], size+1 );
        }
    }
}

printf (GREEN"Final result\n");
for (i = 0; i < cnt; i = i + 1)
{
    if ( array[i][0] == '\0' )
        continue;
    printf ("%s\n", array [i]);
}
printf ("SUCCESS!"WHITE"\n");
free (cubeSharp);
free (array);
free (list);
free (objvArgs);
```

End of presentation.

Thank you!