Module d'approfondissement
Logique, mathématiques,informatique

Eric Abouaf

15 juin 2005

Etude d'un interpréteur Forth

Année Scolaire 2004-2005

Résumé

Ce document est le rapport d'un mini-projet réalisé pour un cours d'approfondissement. Il s'agit de l'implémentation d'un interpréteur pour un langage proche du Forth. Le programme est écrit en C et les sources sont disponibles à la fin de ce document. Une petite introduction au langage Forth sera dispensée afin de comprendre la problématique de l'interpreteur.

Table des matières

1 Introduction

1.1 Un mot pour commencer

Ce document a été réalisé pour le cours d'approfondissement "Logique, mathématiques, informatique" dispensé à l'Ecole Centrale Paris par Mr. Pascal Laurent en mai-juin 2005.

Vous pouvez l'utiliser ou le distribuer comme vous le souhaitez, mais merci de conserver l'intégralité du document afin qu'il garde son sens. Ce document n'ayant pas été relu par une personne plus compétente, je ne garantis pas la véracité de l'ensemble du document.

L'adresse originale de ce travail est : http ://www.neyric.com/comp/forth/ Vous pouvez me contacter pour toute remarque ou question à propos de ce document à l'adresse suivante : eric.abouaf@student.ecp.fr

Le langage implanté est en grande partie basé sur la définition du langage Forth de l'American National Standard for Information Systems datant du 24 Mars 1994. [1] Particulièrement les sections 1 à 6, (Les sections 7 à 17 définissent des word sets supplémentaires non pris en compte dans micro forth.) même si le jeu de mots est bien plus limité (pour l'instant !).

1.2 Objectif

Ce document a pour objectif de se familiariser avec la réalisation d'un interpréteur pour un langage informatique afin de comprendre les concepts de bases d'un compilateur.

1.3 Choix du forth

J'ai choisit le langage forth pour plusieurs raisons :

- interpréteur très petit (dictionnaire des core words très succin),
- portabilité du langage (très près du langage machine et pourtant portable sur n'importe quelle "machine virtuelle" forth),
- simplicité du langage et donc de l'implémentation.

1.4 Portée

Ce document définit l'implémentation d'un système forth minimaliste ne répondant pas au format défini par l'ANS. Sa vocation est purement éducative bien que la plupart des éléments présentés peuvent servir de base à la création d'une implémentation complète du langage forth.

Inclusions :
- ce document décrit le langage "Micro-forth",
- ce document décrit l'implémentation d'un système Micro-forth capable de décrire le langage Micro-forth. Les deux noms seront confondus par la suite.
- ce document s'intéresse particulièrement au coeur du système forth et au mode d'execution.

Exclusions :
- ce document ne vise pas à implémenter le langage forth tel que définit par l'ANS. Pour plus d'informations, sur le langage forth, je vous dirige vers le site de l'ANS [1],
- pour toutes les exclusions du langage forth dans Micro-forth, je vous invite à vous rendre partie 3.1 pour une lire la simplification du langage.

1.5 Organisation du document

Ce document va présenter le cheminement pour parvenir à l'implémentation d'un système forth simplifiée. Le but est de montrer quels éléments du langage permettent d'expliquer les choix réalisés lors de l'implémentation.

2 Introduction au langage Forth

Si vous êtes déjà familiarisé avec le forth, vous pouvez sauter cette partie.

2.1 Introduction

Le Forth est un langage avec une syntaxe assez déroutante mais très efficace. C'est un langage fonctionnel qui se base sur une pile de donnée pour passer les arguments aux fonctions. Ainsi, les fonctions sont représentés par des simples mots, tout comme les variables, les constantes ou les opérateurs de structures.

2.2 La Pile de données

La pile de données est une pile LIFO (Last Input First Output) qui stocke les paramètres d'appel d'un mot. Exemple :

```
2 3 + .
```

Les nombres 2 puis 3 sont empilés puis le mot '+' enlève ces deux nombres de la pile de données et place le résultat de l'opération au sommet de la pile. Le plus effectue l'addition puis le '.' affiche le résultat. On utilise généralement

une notation bien pratique pour décrire l'action d'un mot sur la pile :

```
Mot ( before -- after )
```

Pour notre exemple précédent :

```
+ ( a b -- a+b )
```

Il existe beaucoup de mots de manipulations de la pile. Citons parmi eux :

- DUP : duplique le sommet de la pile
- DROP : enlève l'élément au sommet de la pile
- OVER : duplique l'avant-dernier élément de la pile
- etc...

2.3 Notions de mot et de dictionnaire

L'ensemble des mots forth forme un **dictionnaire**. Il existe deux types de mots :

- les mots de base du forth (appellés core-words)
- les mots définis par l'utilisateur

Pour définir un nouveau mot, on utilise la syntaxe suivante :

```
: nouveau_mot definition ;
```

Voici quelques exemples de définitions :

```
: carre dup * ; ( a -- a^2)
( Calcul le carre d'un nombre )

: cube dup carre * ; ( a -- a^3)
( Calcul le cube d'un nombre )
```

Certains mots de base peuvent également être définis à l'aide d'autres mots (swap échange les deux nombres au sommet de la pile, rot effectue une rotation entre les trois premiers éléments) :

```
: over swap dup rot rot ;
```

2.4 Les variables

Les variables sont également des mots. Lors de leur execution, ces mot mettent sur la pile l'adresse de l'espace mémoire réservé pour cette variable.

Deux mots deviennent alors fondamentaux :

```
@ "at" ( addr -- value )
Ajoute sur la pile le contenu de addr.

! "store" ( value addr -- )
Ecrit value à l'adresse addr.
```

Voici un petit exemple pour illustrer qui illustre la création, l'affectation et la lecture d'une variable :

```
variable test
5 test !
test @ .
```

3 Définition du Micro-forth

3.1 Simplifications

Le but du micro-forth n'est pas de concevoir un nouveau langage révolutionnaire ni d'étudier ses optimisations. L'implémentation souffira donc de nombreuses pertes de vitesse par rapport au langage forth dans un souci de temps. Voici

une liste exhaustive de toute les simplifications :

- liste des core-words très simplifiée,
- calculs sur 32 bits uniquement (plus facile pour l'adressage),
- mots du core défini en forth (perte de performance originellement gagné en jouant sur un jeu d'instructions plus élevé, exemple : 1+ correspond normalement à "inc" alors que 1 + effectue deux instructions pour le processeur),
- nombre non-signés uniquement,
- interpréteur case-sensitive,
- base 10 pour l'affichage de toutes les valeurs.

3.2 Eléments repris du forth

La plupart des éléments fondamentaux du forth sont repris, en l'occurence :

- la syntaxe,
- les mots de manipulations de pile,
- la pile de retour (cf. 4.2.2),
- le buffer d'entrée,
- la structure du dictionnaire.

Tous ces éléments sont accessibles par l'utilisateur et forment également le "coeur" du système forth, à l'aide d'un algorithme relativement simple décrit dans la prochaine partie.

4 Implémentation d'un système Micro-forth

4.1 Choix dans l'implémentation

Voici à nouveau quelques simplifications, cette fois-ci dans l'environnement forth :

- aucune variable d'environnement forth,
- aucun élément de stockage (plus tard, rajouter la lecture de fichier).

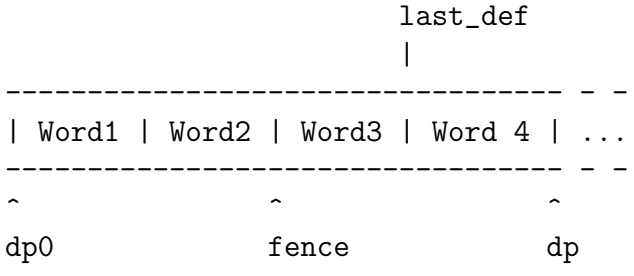
La sortie standard sera toujours l'écran, et l'entrée standard sera le clavier.

4.2 Eléments d'un interpréteur

Nous allons détailler dans cette partie toutes les sous-fonctions d'un interpréteur forth. Ensuite, nous étudierons le fonctionnement conjugué de toutes ces sous-parties.

4.2.1 Structure du dictionnaire

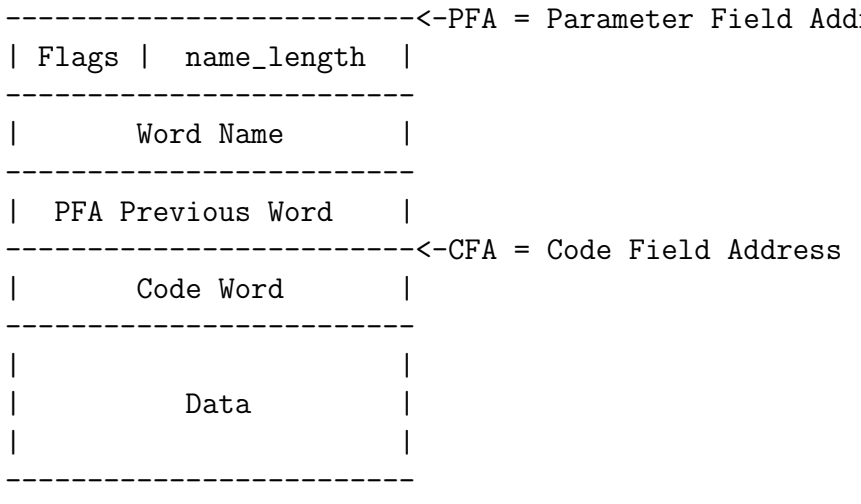
Le dictionnaire contient les définitions de l'ensemble des mots du langage. C'est une zone mémoire réservée lors de l'initialisation du système. Le dictionnaire a la structure suivante :



Le dictionnaire contient 4 pointeurs essentiels à l'éxecution ou à la compilation de nouveaux mots :

- dp0 : pointeur vers le début du dictionnaire,
- dp : pointeur vers la première cellule libre du dictionnaire,
- fence : non utilisé par Micro-forth (cf. [2] pour plus d'informations),
- last_def : adresse du dernier mot définit.

Chaque mot a alors la structure suivante :



- flags : indique si le mot est de type IMMEDIATE ou non (cf. 4.3),
- name lengh : longueur du nom du mot,
- word name : chaîne de charactères représentant le mot (null-terminated),
- pfa previous word : adresse du champ de paramètre de la définition précédente (sert au parcours du dictionnaire lors de la recherche d'un mot),
- code word : indique le mot qui servira à traduire data (le mot peut être un mot du core, une definition, une variable ou une constante),
- data : contient la définition du mot ou la valeur d'une variable.

Plus précisement, dans le cas d'une définition, le champ data comporte la liste des cfa des mots qui le composent.
Pour reprendre notre exemple précédent, le champ data de "carre" contient le cfa de dup suivit du cfa de *.

4.2.2 La pile de données

La pile de données est l'élément le plus simple du forth. Nous avons déjà vu son fonctionnement (cf. 2.2). Son implémentation consiste à allouer l'espace nécessaire. Deux poiteurs sont nécessaires pour manipuler cette pile :

- sp0 : pointeur vers le début de la pile de données,
- sp : poiteur vers le prochain emplacement libre sur la pile.

Le nombre d'éléments sur la pile peut alors être calculé par (sp-sp0)/4 (on divise par 4 car tous les éléments de la pile font 32 bits).

4.2.3 La pile de retour

La pile de retour est un élément essentiel du coeur de forth. Elle contient les adresses de retour lors de l'execution de mot. L'utilisteur peut y accéder

par l'intermédiaire des mots ¿R et R¿ qui transferent le sommet entre la pile de données et la pile de retour. Cependant, si le système trouve une valeur qui n'est pas une adresse de retour au sommet de la pile, l'interpréteur risque de planter. Leur utilisation requiert donc de nombreuses précautions :

- un programme ne doit pas accéder aux valeurs de la pile de retour par R¿ s'il n'a pas placé au préalable une valeur par ¿R,
- un programme ne doit pas accéder aux valeurs de la pile de retour placées avant une boucle depuis l'interieur de cette boucle,
- toutes les valeurs stockés sur la pile de retour dans une boucle doivent être otées avant la fin de la boucle,
- toutes les valeurs placées par ¿R doivent être retirées avant la fin de l'execution de l'input buffer.

4.2.4 Le buffer d'entrée (input buffer)

Le buffer d'entrée est un espace servant de stockage à la chaîne de charactères entrée par l'utilisateur. Cette chaîne sera analysée, découpée en mots

qui seront executés. Une variable ¿IN permet de savoir la position du parser lors de l'interprétation.

4.3 Structure de l'interpréteur

4.3.1 Mots immédiats

Les mots immédiats sont un peu particuliers et servent lors de la compilation de nouveaux mots. Ils modifient le flux de traitement du buffer d'entrée. Exemple : le mot " :" est un mot immédiat et executé à la compilation. Ce

mot va chercher le prochain mot dans l'input buffer et crée une entête dans le dictionnaire. Il passe également le mode de l'interpréteur de exécution à compilation du nouveau mot. Un mot immédiat situé à l'interieur d'une

définition ne sera pas compilé dans cette définition. Ce sont de véritables "agents de contrôle" de la compilation.

4.3.2 Registre IP (instruction pointer)

Le registre IP contient l'adresse du prochain mot à exécuter. Dans le cas de notre exemple, le système effectuera la tâche suivante : : carre dup *; 1.

Place le cfa de carre sur la pile de retour.
2. IP prend la valeur du premier élément sur la pile de retour.
3. Ajout de IP+4 sur la pile de retour.
4. Execution de [IP] (ici dup).
5. Dépile un élément de la pile de retour dans IP.
6. Execution de [IP] (*).
7. Plus rien sur la pile de retour, donc on rend la main.

4.3.3 Fonctionnement interne

```
I) Récupère le prochain mot du buffer d'entrée
II) Chercher le mot dans le dictionnaire
    1. Si il est trouvé
        i) Si le mode est COMPILATION:
            - Execute le mot s'il est marqué IMMEDIATE
            - Sinon, ajouter son cfa à la suite de la
             définition du dictionnaire
        ii) Si le mode est INTERPRETATION:
            - Execute le mot
    2. S'il n'est pas trouvé
        i) Si c'est un nombre
            - Ajouter à la pile de données en INTERPRETATION
            - Ajouter à la définition, précédé du mot (VALUE)
             en mode COMPILATION
        ii) Sinon, affiche un message d'erreur "Word unknown"
```

5 Conclusion

Grâce à une structure ingénieuse, le forth est un langage nécessitant très peu de ressources. Il est de plus extrement rapide puisqu'une fois compilé, l'exécution ne fait qu'une série de "jump" de plus que le code assembleur lui-même. Finalement, une fois familiarisé avec la structure interne d'un interpréteur forth, on se rend compte que celui-ci fonctionne comme un ensemble de **règles de Markov** pour du code executable. En effet, lors de l'execution d'un mot, on ne fait que remplacer ce mot par sa définition jusqu'à tomber sur une succession de mots de base directement executables. L'interpréteur présenté en annexe n'est pas encore complètement fonctionnel puisque le vocabulaire de base est très limité. Cependant, beaucoup de mot peuvent se définir à l'aide de la pile de retour, comme le do ... loop qui sert à faire les boucles :

```
: do
   compile swap
   compile >r
   compile >r
   here ; immediate

: loop
   compile r>
   compile 1+
   compile r>
   compile 2dup
   compile >r
   compile >r
   compile <
   compile 0=
   compile 0branch
   here - ,
   compile r>
   compile r>
   compile 2drop  ; immediate
```

Annexes

A Bibliographie

[1] http ://www.taygeta.com/forth/dpans.html
[2] http ://forth.free.fr/livres/guide/guide.htm

B Code Source de Microforth Interpreter

Voici le code source de l'interpreteur.
Il peut être téléchargé sous forme informatique à l'adresse suivante :
http ://www.neyric.com/comp/forth/microforth.tar.gz

```
:::::::::::::::
main.c
:::::::::::::::
/*
     main.c
     Microforth interpreter

     By Eric Abouaf
     neyric@via.ecp.fr
     June 10, 2005

     This file contains the main initialisations and the infinite loop.

*/

#include <stdio.h>          // printf
#include <stdlib.h>         // exit

#include "main.h"

#include "dictionnary.h"
#include "stacks.h"
#include "input_buffer.h"
#include "parser.h"


// init()
//
// Function:    Prints infos
//              Allocates space for stacks and dictionnary
// Arguments:   None
// Return:      Nothing
//
void init()
{
     // Prints infos
     printf("Microforth interpreter\n");
     printf("By Eric Abouaf\n");
     printf("neyric@via.ecp.fr\n");
     printf("June, 2005\n\n");

     // Allocates space for input_buffer, dictionnary, stacks
     init_inputbuffer();
     init_dictionnary();
     init_stacks();
}

// free_mem()
//
// Function:    Frees the allocated memory
// Arguments:   None
// Return:      Nothing
//
void free_mem()
```

```
{
    // Free allocated memory for dictionnary, input_buffer, stacks
    free_inputbuffer();
    free_dictionnary();
    free_stacks();
}


// main()
//
// Function:    Entry point of the application
// Parameters:  None
// Return:      0 (always)
//
int main(void)
{
   // Allocates memory
   init();

   // Infinite loop
   for( ; ; )
   {
        // Print prompt
        printf("␣OK\n");

        // Fills input_buffer
        fill_inputbuffer();

        // Call the parser
        parser();
   }

   // Never called, for compilation
   return 0;
}
```

```
:::::::::::::::
main.h
:::::::::::::::

void free_mem();
```

```
:::::::::::::::
dictionnary.c
:::::::::::::::
/*

    dictionnary.c
    Microforth interpreter

    By Eric Abouaf
    neyric@via.ecp.fr
    June 10, 2005

    This file contains the main dictionnary variables
    ans allocate space for the dictionnary.

    The structure of the dictionnary is the following :

                                last_def
                                |
    ------------------------------------ - -
    | Word1 | Word2 | Word3 | Word 4 | ...
    ------------------------------------ - -
    ^                ^                ^
    dp0              fence            dp

    dp0:          start of the dictionnary
    dp:           first cell available at the end of the dictionnary
    fence:        before fence, the interpreter cannot FORGET
    last_def:     Adress of the last definition

    The structure of a word within the dictionnary is the following:

PFA ----------------------
    | Flags |  name_length  |
    ------------------------
    |        Word Name      |
    ------------------------
    |     Previous Word     |   Pointer to the PFA of the previous word
CFA ----------------------
    |        Code Word      |        EXEC_WORD | COLON_WORD
    ------------------------
```

```
    |                       |
    |          Data         |
    |                       |
    -------------------------
*/

#include "dictionnary.h"
#include "core.h"

#define     DICO_SIZE       8000


#define     EXEC_WORD       0
#define     COLON_WORD      1

#define     IMMEDIATE_WORD  32

void * dp0;
void * dp;
void * fence;
void * last_def;

// init_dictionnary()
//
// Function:    Allocates memory for the dictionnary
//              and initiate dictionnary pointers
// Arguments:   None
// Returns:     Nothing
//
void init_dictionnary()
{
    // Allocates memory
    // +1 => no definition starting by 0
    dp0 = (void *) malloc(DICO_SIZE)+1;

    // Set dico pointers
    dp = dp0;
    fence = dp0;

    // First definition has no previous word
    last_def = 0;

    core_definitions();
}

// core_definitions()
//
// Function:    Creates entries in the dictionnary
// Arguments:   None
// Returns:     Nothing
//
void core_definitions()
{
     add_core_word(".", &dot, 0);
     add_core_word(".s", &dots, 0);
     add_core_word("+", &plus, 0);
     add_core_word("words", &words, 0);
     add_core_word("quit", &quit, 0);
     add_core_word(":", &colon, IMMEDIATE_WORD);
     add_core_word(";", &semicolon, IMMEDIATE_WORD);
     add_core_word("dp@", &dpat, 0);
     add_core_word("dup", &dup, 0);
     add_core_word("*", &mult, 0);
}

// free_dictionnary()
//
// Function:    Frees the memory allocated for the dictionnary
// Arguments:   None
// Returns:     Nothing
//
void free_dictionnary()
{
    // frees the memory of the dictionnary
    free(dp0);
}


// add_core_word()
//
// Function:    Create a new entry in the dictionnary
// Arguments:
//      - word:     string of the word
```

```
//      - function: adress of the C function
//      - flags:    IMMEDIATE or NULL
// Returns:     Nothing
//
void add_core_word(const char *word, void *function, unsigned char flags)
{
    void * temp_dp = dp;

    unsigned char temp;

    // Get the size of the word (between 1..31)
    unsigned char length = (unsigned char) strlen(word);
    unsigned char length_flags = length | flags;

    // Write the first byte (flags+length)
    memcpy(dp, &length_flags, sizeof(unsigned char) );
    dp += sizeof(unsigned char);

    // Write the name of the word
    // adds the zero at the end for strcmp
    memcpy(dp, (void *) word, length+1);
    dp += sizeof(char)*(length+1);

    // Write the pfa of the previous word
    memcpy(dp, &last_def, sizeof(void *) );
    dp += sizeof(void *);

    // Write the Code Word, which is in this case only EXEC_WORD
    temp = EXEC_WORD;
    memcpy(dp, &temp, sizeof(unsigned char) );
    dp += sizeof(unsigned char);

    // Write the PFA (Address of the execution function)
    memcpy(dp, &function, sizeof(void *) );
    dp += sizeof(void *);

    // Set the last_def variable
    last_def = temp_dp;
}

:::::::::::::::
dictionnary.h
:::::::::::::::

// Contains definitions for dictionnary.c

void init_dictionnary();
void core_definitions();
void free_dictionnary();
void add_core_word(const char *word, void *function, unsigned char flags);

:::::::::::::::
input_buffer.c
:::::::::::::::
/*
    input_buffer.c
    Microforth interpreter

    By Eric Abouaf
    neyric@via.ecp.fr
    June 10, 2005

    This file contains the input buffer functions: init, free, and fill

*/

#include "input_buffer.h"

#define INPUTBUFFER_SIZE    256

// Input buffer pointer
char * input_buffer;

// init_inputbuffer
//
// Function:    Allocate memory for the input_buffer
// Arguments:   None
// Return:      Nothing
//
void init_inputbuffer()
{
    // Allocates space for input_buffer
    input_buffer = (char *) malloc(INPUTBUFFER_SIZE);
```

```
    if(input_buffer == 0)
    {
        printf("init_inputbuffer: Unable to allocate space for input_buffer\n");
    }
}

// free_inputbuffer
//
// Function:     Free memory allocated for the input_buffer
// Arguments:    None
// Return:       Nothing
//
void free_inputbuffer()
{
   // frees input_buffer
   free( (void *) input_buffer);
}

// fill_inputbuffer
//
// Function:     Fills the input_buffer with data from stdin
//               and prevent buffer overflow. (better than gets)
// Arguments:    None
// Return:       Nothing
//
void fill_inputbuffer()
{
   // Fills input_buffer
   char c = 0;
   unsigned char i = 0;

   while( c != '\n' )
   {
       // Get char from stdin
       c = getchar();

       // Prevent buffer overflow
       if( i == INPUTBUFFER_SIZE-1)
           c = '\n';

       // Add the letter to the buffer
       input_buffer[i++] = c;
   }
   input_buffer[i-1] = 0;     // i-1 cause we remove the '\n'
}

:::::::::::::::
input_buffer.h
:::::::::::::::

// contains definitions for input_buffer.c

void init_inputbuffer();
void free_inputbuffer();
void fill_inputbuffer();

:::::::::::::::
parser.c
:::::::::::::::
/*
    parser.c
    Microforth interpreter

    By Eric Abouaf
    neyric@via.ecp.fr
    June 10, 2005

    This file parse the input buffer and execute the following:

    I) Parse next word in input buffer
    II) Lookup the word in the dictionnary
        1. If found
            i)if compiling:
                - Execute if the word is marked IMMEDIATE
                - Else add word at the end of the dictionnary
            ii)if interpreting:
                - Execute the word
        2. If not found
            i) Try to convert it to a number
                - add to stack if interpreting
                - add to dictionnary with (VALUE) before in the dictionnary
                    if compiling.
            ii) Print an error message "Word unknown"
```

```
*/

#include "parser.h"

#define     MODE_COMPILATION    1
#define     MODE_INTERPRET      0

extern char * input_buffer;
extern void * last_def;
extern void * dp0;
extern void * dp;
extern void * sp;

extern void * rp;
extern void * rp0;

char * in;

unsigned char mode = MODE_INTERPRET;

// GetNextWord()
//
// Function:    Gets next word in input buffer
// Arguments:   next_word => result buffer
// Returns:     Nothing
//
void GetNextWord( char * next_word)
{
    int i = 0;

    // Remove blanks or tabs before the word
    while( in[0] == ' ' || in[0] == '\t' )
        in += sizeof(char);

    // Store the word
    while( in[0] != ' ' && in[0] != '\t' && in[0] != 0 )
    {
        next_word[i++] = in[0];
        in += sizeof(char);
    }
    next_word[i++] = 0;
}

// tick()
//
// Function:    find a word given its name in the dictionnary
// Arguments:   word: string of the word
// Returns:     pointer containing the pfa to the word
//
void * tick(unsigned char *word)
{
    // Take the last_definition
    void * definition = last_def;
    void * pfa = 0;
    unsigned char boolean = 0;

    // While we didn't find the word and no more definition
    while( boolean == 0 && definition != 0)
    {
        void * name_ptr = definition+sizeof(unsigned char);

        // Compare the name
        if( strcmp(word, (unsigned char *) name_ptr) == 0 )
        {
            pfa = definition;
            boolean = 1;
        }
        else    // If different, go to previous definition
            memcpy( &definition, name_ptr+strlen((unsigned char *) name_ptr)+1, sizeof(void *) );
    }

    return (void *) pfa;
}

// pfa2cfa()
//
// Function:    convert a pfa to a cfa
// Arguments:   pfa of the word
// Returns:     pointer containing the cfa of the word
//
void * pfa2cfa(void * pfa)
{
```

```
    void * cfa;
    unsigned char length;

    if( pfa == 0) return 0;

    // Skip length byte
    cfa = pfa + sizeof(unsigned char);

    // Skip name
    memcpy(&length,pfa,sizeof(unsigned char));
    length = length & 31;

    cfa += (length+1)*sizeof(char);

    // Skips previous pfa
    cfa += sizeof(void *);

    return cfa;
}

// is_numeric()
//
// Function:     Test if a string is numerical
// Arguments:    string
// Returns:      0 if numerical, else 1.
//
char is_numeric(unsigned char * str)
{
     unsigned char * temp = str;
     while(temp[0] != 0 )
     {
        if( temp[0] < '0' || temp[0] > '9' )
             return 1;

        temp++;
     }
     return 0;
}


// parser()
//
// Function:     See top of the file
// Arguments:    None
// Return:       Nothing
//
void parser()
{
    // To store the next word
    char next_word[32];

    void * cfa;
    int length = strlen(input_buffer);

    // Makes in the beggining of input_buffer
    in = input_buffer;

    // Continues until the input_buffer gets empty
    while( in != input_buffer+length * sizeof(char) )
    {
        GetNextWord(next_word);

        // If word is not empty...
        if( next_word[0] != 0 )
        {
            // Tick find the pfa of the word
            unsigned char firstbyte = 0;
            void * temp_pfa = tick(next_word);


            cfa = pfa2cfa( temp_pfa );

            if( cfa != 0)
            {
                memcpy( &firstbyte, temp_pfa, sizeof(unsigned char) );
                // & 32 => Filter for the 3rd bit
                do_real_word(cfa, firstbyte & 32);
            }
            // Else cfa == 0
            else
            {
                // Try to convert to a numerical value
                if( is_numeric(next_word) == 0)
```

```
            {
                long number = atoi(&next_word);

                if ( mode == 0)  // INTERPRETATION
                {
                    // Add the number on stack
                    memcpy(sp,&number,sizeof(long) );
                    sp += sizeof(long);
                }
                else
                {
                    // Add the CFA of value
                    void * value = (void *) 1;
                    memcpy(dp,&value,sizeof(void *) );
                    dp += sizeof(void *);

                    // Add the value
                    memcpy(dp,&number,sizeof(long) );
                    dp += sizeof(long);
                }
            }
            else  // Word unknown
                printf("%s: Undefined word\n", next_word);
        }  // else cfa == 0

    } // If word not empty
  }    //while in...

}

// do_real_word()
//
// Function:    See top of the file
// Arguments:   cfa of the word to execute or compile
//              flags ( immediate word or not )
// Return:      Nothing
//
void do_real_word(void * cfa, unsigned char flags)
{
    if( mode == 1) // COMPILATION
    {
        // IMMEDIATE WORD
        if( flags == 32)
            interpret(cfa);
        else
        {
            // We just add the cfa at the end of the dictionnary
            memcpy(dp, &cfa, sizeof(void *) );
            dp += sizeof(void *);
        }
    }
    else     // INTERPRETEATION
        interpret(cfa);
}

// interpret()
//
// Function:    Execute a word given its cfa
// Arguments:   cfa
// Return:      Nothing
//
void interpret( void * cfa )
{
    unsigned char type;
    void (*fp)();
    void * cfa_data = cfa + sizeof(unsigned char);

    memcpy(&type, cfa, sizeof(unsigned char) );

    if( type == 0) // EXEC_WORD
    {
       // Retrieve exec address
       memcpy( &fp, cfa_data, sizeof(void *) );
       // Execute address
       fp();
    }
    else if (type == 1) // COLON_WORD
    {
       // Add the cfa on top of the return stack
       memcpy(rp, &cfa_data , sizeof(void *) );
       rp += sizeof(void *);

       printf("", cfa_data);
```

```
        // Execute the stack
        execute();
    }
    else
        printf("Corrupted memory in dictionnary.\n");
}

// execute()
//
// Function:     Execute the return stack (if forth-defined word)
// Arguments:    None
// Return:       Nothing
//
void execute()
{
    void * cfa;
    void * ip;

    long temp;

    while( rp != rp0)
    {
        // Unstack an element
        rp -= sizeof(void *);
        memcpy(rp, &cfa, sizeof(void *) );

        ip = cfa;
        // printf("%d ", cfa);

        memcpy(&temp, ip, sizeof(long) );
        while(temp != 0)
        {
            if( temp == 1)   // 1 is the CFA indicating a value
            {
                ip += sizeof(long *);

                // Add the value on the stack
                memcpy(sp,ip,sizeof(long) );
                sp += sizeof(long);
            }
            else
            {
              void * new_cfa;
              memcpy( &new_cfa, ip, sizeof(void *) );

              interpret(new_cfa);
            }

            // Go to next instruction
            ip += sizeof(long *);
            memcpy(&temp, ip, sizeof(long) );
        }

    }

}


:::::::::::::::
parser.h
:::::::::::::::

// Definitions for parser.c

void GetNextWord( char * next_word);
void parser();
void do_real_word(void * cfa, unsigned char flags);
void interpret( void * cfa);
void execute();

:::::::::::::::
stacks.c
:::::::::::::::
/*
    stacks.c
    Microforth interpreter

    By Eric Abouaf
    neyric@via.ecp.fr
    June 10, 2005

    Contains the init and free functions for data and return stacks.
```

```
*/

#include "stacks.h"


#define     DATASTACK_SIZE   3000
#define     RETURNSTACK_SIZE   3000


// Pointers for the return stack
void * rp0;
void * rp;

// Pointers for the data stack
void * sp0;
void * sp;

// init_stacks()
//
// Function:     Allocate space for data and return stacks
// Arguments:    None
// Return:       Nothing
//
void init_stacks()
{
    // Allocates space for data stack
    sp0 = (void *) malloc(DATASTACK_SIZE);
    if(sp0 == 0)
    {
        printf("init_stacks: Unable to allocate space for data stack\n");
    }
    sp = sp0;

    // Allocates space for return stack
    rp0 = (void *) malloc(RETURNSTACK_SIZE);
    if(rp0 == 0)
    {
        printf("init_stacks: Unable to allocate space for return stack\n");
    }
    rp = rp0;
}

// free_stacks()
//
// Function:     Free memory allocated for the stacks
// Arguments:    None
// Return:       Nothing
//
void free_stacks()
{
   // Free stacks
   free( (void *) rp0);
   free( (void *) sp0);
}

:::::::::::::::
stacks.h
:::::::::::::::

// definitions for stacks.c

void init_stacks();
void free_stacks();


:::::::::::::::
core.c
:::::::::::::::
/*
    core.c
    Microforth interpreter

    By Eric Abouaf
    neyric@via.ecp.fr
    June 11, 2005

    This file contains some core word definitions.
    All the functions are meant to be added to the dictionnary
    through add_core_word()
*/
```

```
#include "core.h"

#include "parser.h"
#include "main.h"

extern void * sp;
extern void * dp;
extern void * sp0;
extern void * last_def;

extern unsigned char mode;

// dup
void dup()
{
    if(sp == sp0)
        printf("dup: error, empty stack\n");
    else
    {
        memcpy(sp, sp-sizeof(long), sizeof(long) );
        sp += sizeof(long);
    }
}

// DP@
void dpat()
{
    memcpy(sp, &dp, sizeof(void *) );
    sp += sizeof(void *);
}

// : (immediate word)
void colon()
{
    char next_word[32];
    unsigned char length;
    unsigned char temp;

    void * temp_dp = dp;

    // Switch to compilation mode
    mode = 1;

    // GetNextWord in input_buffer
    GetNextWord(next_word);
    length = (unsigned char) strlen(next_word);

    // write word header
    memcpy(dp, &length, sizeof(unsigned char) );
    dp += sizeof(unsigned char);

    // write the name
    memcpy(dp, (void *) next_word, length+1);
    dp += sizeof(char)*(length+1);

    // Write the pfa of the previous word
    memcpy(dp, &last_def, sizeof(void *));
    dp += sizeof(void *);

    // Write info about a colon word
    temp = 1;
    memcpy(dp, &temp, sizeof(unsigned char) );
    dp += sizeof(unsigned char);

    // Update last_def
    last_def = temp_dp;
}

// ; (immediate word)
void semicolon()
{
    void * temp = 0;

    // Switch to interpretation mode
    mode = 0;

    // Write the end of the word (0)
    memcpy(dp, &temp, sizeof(void *) );
    dp += sizeof(void *);
}

// quit
void quit()
```

```
{
    printf("\nGoodbye !\n\n");
    free_mem();
    exit(0);
}

// .
void dot()
{
     if(sp == sp0)
          printf(".: error, empty stack\n");
     else
     {
          long temp;
          sp -= sizeof(long);
          memcpy(&temp, sp, sizeof(long) );
          printf("%d ", temp);
     }
}

// +
void plus()
{
     if(sp <= sp0+sizeof(long) )
          printf("+: error, empty stack\n");
     else
     {
          long x1, x2;
          sp -= sizeof(long);
          memcpy(&x1, sp, sizeof(long) );
          sp -= sizeof(long);
          memcpy(&x2, sp, sizeof(long) );

          x1 += x2;
          memcpy(sp, &x1, sizeof(long) );
          sp += sizeof(long);
     }
}

// *
void mult()
{
     if(sp <= sp0+sizeof(long) )
          printf("+: error, empty stack\n");
     else
     {
          long x1, x2;
          sp -= sizeof(long);
          memcpy(&x1, sp, sizeof(long) );
          sp -= sizeof(long);
          memcpy(&x2, sp, sizeof(long) );

          x1 = x1*x2;
          memcpy(sp, &x1, sizeof(long) );
          sp += sizeof(long);
     }
}

// .s
void dots()
{
     void * temp = sp0;

     printf("<%d> ", (sp-sp0)/sizeof(long));

     while( temp != sp)
     {
          long numero;
          memcpy(&numero, temp, sizeof(long) );
          printf("%d ", numero);
          temp += sizeof(long);
     }
}


// words
void words()
{
     void * definition = last_def;

     // While we didn't find no more definition
     while( definition != 0)
     {
```

```
        void * name_ptr = definition+sizeof(unsigned char);

        printf("%s ", (unsigned char *) name_ptr);

        // Previous definition address is at name_ptr+lengthof(name)
        memcpy( &definition, name_ptr+strlen((unsigned char *) name_ptr)+1, sizeof(void *) );
    }
    printf("\n");
}


:::::::::::::::
core.h
:::::::::::::::

// Contains declarations of core.c

void dup();
void mult();
void dpat();
void colon();
void semicolon();
void quit();
void dot();
void dots();
void plus();
void words();

:::::::::::::::
Makefile
:::::::::::::::
microforth: main.c dictionnary.c input_buffer.c stacks.c parser.c core.c
        gcc -o main.o -c main.c
        gcc -o dictionnary.o -c dictionnary.c
        gcc -o core.o -c core.c
        gcc -o parser.o -c parser.c
        gcc -o input_buffer.o -c input_buffer.c
        gcc -o stacks.o -c stacks.c
        gcc -o microforth main.o dictionnary.o input_buffer.o stacks.o parser.o core.o

clean:
        rm -rf *.o
        rm microforth
```