Construction et évaluation d'arbres syntaxiques abstraits

Ensimag 1A - Préparation au Projet C

Présentation

L'objet de cet exercice est de réaliser un ensemble de sous-programmes permettant de construire des *patchworks* [1] composés d'images *primitives*. Les patchworks considérés sont inscrits chacun dans un rectangle dont les côtés sont parallèles aux directions principales de la surface d'affichage. Ils sont construits à partir de deux images *primitives* (cf. figure 1) au moyen d'opérateurs de composition et de transformation. Des exemples sont représentés sur la figure 2.

FIGURE 1 – Les deux images primitives

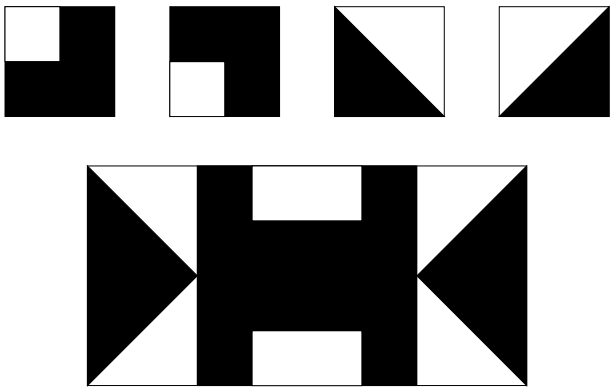

FIGURE 2 – Exemples de patchworks composés à partir des images primitives

On peut définir l'ensemble des patchworks considérés par les 5 règles suivantes :

1. les deux images primitives sont des patchworks ;
2. si on juxtapose horizontalement deux patchworks de même hauteur, on obtient un patchwork ;
3. si on superpose deux patchworks de même largeur, on obtient un patchwork ;
4. si on applique à un patchwork une rotation de 90 degrés dans le sens trigonométrique, on obtient un patchwork ;
5. il n'y a pas d'autre patchwork que ceux que l'on peut obtenir à partir de deux images primitives par l'application d'un nombre fini de juxtapositions, de superpositions et de rotations.

1. Ce sujet s'inspire d'un exemple tiré de l'ouvrage *Programming Languages : Concepts and Constructs* de Ravi Sethi.

Une image primitive est toujours associée à une orientation (EST par défaut) modifiée par l'opération de rotation.

Du point de vue de l'utilisateur final, un patchwork est décrit au moyen d'un langage d'expressions basé sur des mots-clés désignant les images primitives (**`carre`** et **`triangle`**), sur des symboles associés aux opérateurs utilisables (par exemple : **`#`** pour la juxtaposition, **`/`** pour la superposition et **`@`** pour la rotation) et enfin sur la possibilité de parenthéser des sous-expressions. Une expression permettant de décrire dans ce langage le patchwork le plus complexe de la figure 2 serait :

```
(@@triangle / @triangle) # ((carre # @carre) / @@(carre # @carre)) #
(@@@triangle / triangle)
```

La correction lexicale et syntaxique d'une telle expression doit être analysée au moyen d'un analyseur syntaxique (un "parser") qui, si l'analyse est effectuée avec succès, produira une représentation de l'expression sous la forme d'un *arbre syntaxique abstrait* ("Abstract Syntax Tree", AST en abrégé). Un exemple est donné en figure 3.

Remarque importante : la réalisation d'un analyseur syntaxique du langage dont il est question ci-dessus *n'est pas demandée* dans le cadre de cet exercice.

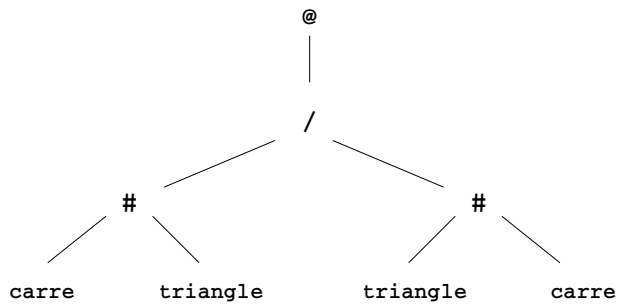

FIGURE 3 – Exemple d'arbre syntaxique abstrait de l'expression **`@((carre # triangle) / (triangle # carre))`**

Une fois construit, l'AST d'une expression est destiné à être parcouru pour générer le patchwork qui lui est associé. Les nœuds de l'AST sont ainsi évalués, des feuilles vers la racine. L'évaluation d'un nœud intermédiaire engendre la génération d'un patchwork *temporaire*, résultat de l'évaluation de l'expression correspondant à ce sous-arbre.

L'évaluation du nœud racine permet ainsi de construire un patchwork à partir duquel pourra être générée l'image correspondant à l'expression initiale. Si l'évaluation d'un nœud engendre l'application d'une opération illicite, comme la juxtaposition de patchworks de hauteurs différentes par exemple, l'expression est invalide et le programme est arrêté.

L'enchaînement des opérations à effectuer pour obtenir une image à partir d'une expression donnée est représenté sur la figure 4 sous la forme d'un diagramme de flot de données.

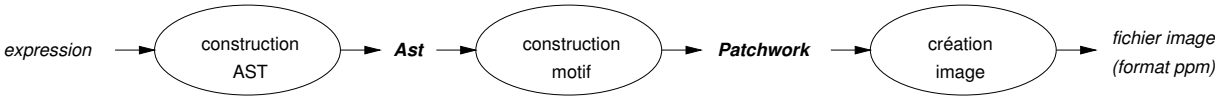

FIGURE 4 – Création d'une image à partir d'une expression

Les sous-programmes réalisant les opérations requises pour obtenir une image sont regroupés en trois modules :

— module `ast` : création et parcours d'un AST ;
— module `patchwork` : création d'un patchwork ;
— module `image` : création d'une image au format `.ppm` à partir d'un patchwork.

L'objectif de ce TP est de compléter les sous-programmes de ces modules. Vous serez amenés à étudier et à manipuler les notions suivantes du langage C et de son environnement :

— structures et unions ;
— pointeurs de fonction ;
— allocation dynamique / libération de la mémoire ;
— manipulation de fichiers binaires.

La compilation devra être faite à partir d'un Makefile écrit par vos soins.

Travail demandé

Module `ast`

Les nœuds d'un arbre syntaxique abstrait sont de deux types : (1) les nœuds intermédiaires qui correspondent chacun à une opération, et (2) les feuilles qui correspondent chacune à une valeur, c'est-à-dire dans notre cas à l'une des images primitives. Une opération est soit binaire (juxtaposition (`#`), superposition (`/`)), soit unaire (rotation (`@`)).
A chaque noeud de l'AST sont associées deux procédures, sous la forme de pointeurs de fonction :

1. une fonction evaluer qui évalue l'expression, c'est-à-dire qui construit le motif correspondant à la description donnée par l'AST ;
2. une procédure afficher qui pourra être utilisée pour afficher les informations contenues dans un nœud et ainsi vérifier visuellement l'intégrité de l'AST.

Ces pointeurs de fonction doivent être initialisés lors de la création de chaque nœud en fonction de son type. Par exemple, si le nœud considéré est de type binaire, on pourra faire pointer afficher vers une fonction afficher_binaire qui affiche un nœud binaire de manière adéquate : affiche le fils gauche, l'information portée par le noeud, le fils droit.

Ainsi, lors du parcours d'un AST, il n'est pas nécessaire d'utiliser d'instruction conditionnelle (**if** ou **switch**) sur le type du nœud pour connaitre la « *bonne* » procédure, il suffit d'exécuter afficher.

Une proposition de fichier d'en-tête pour le module `ast` est disponible en annexe I, page 5.

Module `patchwork`

Ce module contient la structure de données décrivant un patchwork ainsi que les opérations qui lui sont applicables. Un fichier d'en-tête pour ce module vous est proposé en annexe II, page 8).

Module `image`

Ce module permet de créer une représentation graphique d'un patchwork, sous la forme d'un fichier binaire au format PPM binaire (P6) déjà utilisé lors de l'APP 2. Il utilise le module `patchwork` et offre une seule procédure dont le prototype est donné en annexe III, page 9.

Extensions possibles

Cet APP peut être facilement étendu. Nous vous encourageons à discuter des extensions possibles le matin **avant de commencer à programmer** de manière à prendre en compte leur développement dans votre plan d'action de groupe. En particulier, essayez d'analyser la difficulté de vos extensions (temps de développement, code plus ou moins complexe, modifications plus ou moins intrusives, expertise sur le code à modifier, ...) avant de vous lancer dans leur conception.

Quelques extensions possibles (non triées) pour cet APP :

— ajout de nouvelles opérations (inversion des couleurs, symétries axiales, fusion "*moitié-moitié*", ...) ;
— gestion d'images primitives non carrées ;
— interfaçage avec l'APP2 : utilisation de la bibliothèque de motifs pour charger les images primitives ou enregistrer de nouveaux patchworks.

Matériel fourni et consignes générales

Les fichiers suivants vous sont fournis :

— fichiers d'interface (`.h`) des trois modules `ast`, `patchwork` et `image` ;
— fichier `ast.c` contenant les définitions des structures locales et les procédures du module à compléter ;
— programmes de test à compléter ;
— Makefile à compléter.

Vous devrez enfin vous assurer à l'aide du logiciel `valgrind` que les accès mémoire sont valides et que la totalité de la mémoire allouée est libérée à la fin de l'exécution de votre programme.

Un module parser (`parser.h`, `parser.o`) est également distribué. Il fournit une procédure permettant de construire un AST, en utilisant votre propre module AST, à partir d'un fichier texte comprenant une expression. Il servira essentiellement pour des tests d'expressions particulièrement longues.

Annexe I : Module AST

Entête du module `ast.h`

```
#ifndef AST_H
#define AST_H

#include "patchwork.h"

/* Natures des operations sur les motifs */
enum nature_operation {
        ROTATION,
        JUXTAPOSITION,
        SUPERPOSITION,
        NB_OPERATIONS    /* sentinelle */
};


/* Structure de noeud de l'arbre (AST: Abstrat Syntax Tree), representant une
 * expression de type operation unaire ou binaire (noeud interne) ou une
 * valeur (image primitive, sur une feuille).
 * La partie data est "abstraite" et privee, simplement declaree ici et
 * definie dans le fichier .c.
 * Pour la partie "methodes" (afficher et evaluer), on donne simplement le profil
 * generique des fonctions. Les fonctions specifiques a utiliser, qui dependront
 * du type de noeud, seront definies de maniere statique dans le .c. Les champs
 * afficher et evaluer des noeuds devront etre initialises pour pointer vers la
 * bonne fonction lors de la creation de chaque noeud.
 */
struct noeud_ast_data;

struct noeud_ast {
        // donnees privees du noeud
        struct noeud_ast_data *data;

        /* pointeur vers la fonction d'affichage du noeud (specifique du type de noeud):
         * affiche l'expression portee par l'arbre de racine ast, en notation infixee.
         * Pas de verification de la syntaxe: si un noeud est NULL, il n'est
         * simplement pas affiche (ou "null"). */
        void (*afficher) (struct noeud_ast *);

        /* pointeur vers la fonction d'evaluation du noeud (specifique du type de noeud):
         * applique la procedure d'evaluation a l'arbre de racine ast,
         * et retourne le patchwork cree correspondant.
         * En cas d'erreur syntaxique, un message approprie est affiche,
         * toute la memoire liberee et l'execution interrompue. */
        struct patchwork *(*evaluer) (struct noeud_ast *);

        /* on pourrait utiliser le meme mecanisme pour liberer un noeud;
         * ce n'est pas fait ici pour voir les differences entre les deux modeles
         * (regarder liberer_expression par exemple) */
};


/*-------------------------------------------------------------------------*/
```

```
/* Libere la memoire associee a l'arbre ast passe en parametre. */
extern void liberer_expression(struct noeud_ast *ast);

/* Cree et retourne une valeur (feuille d'un AST) correspondant
 * a une image primitive de nature nat_prim. */
extern struct noeud_ast *creer_valeur(const enum nature_primitif nat_prim);

/* Cree et retourne un noeud correspondant a une operation unaire
 * de nature nat_oper, avec opde comme operande. */
extern struct noeud_ast *creer_unaire(const enum nature_operation nat_oper,
                                      struct noeud_ast *opde);

/* Cree et retourne un noeud correspondant a une operation binaire
 * de nature nat_oper, avec opde_g et opde_d comme operandes. */
extern struct noeud_ast *creer_binaire(const enum nature_operation nat_oper,
                                       struct noeud_ast *opde_g,
                                       struct noeud_ast *opde_d);

#endif /* AST_H */
```

Structures locales et pointeurs de fonctions (`ast.c`)

```
/*-------------------------------------------------------------------------*/
/*     FONCTIONS PORTEES PAR LES NOEUDS                                    */
/*  fonctions suivant les signatures des champs afficher et evaluer        */
/*  definies dans ast.h, adaptees aux differentes natures de noeud d'un ast */
/*-------------------------------------------------------------------------*/

/*------------ Affichage */
/* Fonctions "specifiques" locales (static) d'affichage d'un noeud
   selon sa nature. Declarees ici, definies plus bas */
static void afficher_valeur(struct noeud_ast *ast);
static void afficher_unaire(struct noeud_ast *ast);
static void afficher_binaire(struct noeud_ast *ast);


/*------------ Evaluation */
/* Fonctions "specifiques" locales (static) d'evaluation d'un noeud
   selon son type. Declarees ici, definies plus bas */
static struct patchwork *evaluer_valeur(struct noeud_ast *ast);
static struct patchwork *evaluer_unaire(struct noeud_ast *ast);
static struct patchwork *evaluer_binaire(struct noeud_ast *ast);


/*------------ Creation des patchworks */

/* Types des pointeurs sur les fonctions de creation des patchworks.
 * Les signatures different selon les noeuds.
 * Les fonctions specifiques sont definies ds le module patchwork.o */
typedef struct patchwork *(*creer_patchwork_valeur_fct)
                                        (const enum nature_primitif);
typedef struct patchwork *(*creer_patchwork_unaire_fct)
                                        (const struct patchwork *);
typedef struct patchwork *(*creer_patchwork_binaire_fct)
                                        (const struct patchwork *,
                                         const struct patchwork *);
```

```
/*-------------------------------------------------------------------------*/
/*     STRUCTURES DES NOEUDS                                               */
/*-------------------------------------------------------------------------*/

enum nature_noeud {
        VALEUR,          /* feuille */
        OPERATION,       /* noeud interne */
        NB_NAT_NOEUDS    /* sentinelle */
};

enum arite_operation {
        UNAIRE,
        BINAIRE,
        NB_ARITES_OPERATIONS    /* sentinelle */
};


struct operation_unaire {
        struct noeud_ast *operande;
        creer_patchwork_unaire_fct creer_patchwork;
};

struct operation_binaire {
        struct noeud_ast *operande_gauche;
        struct noeud_ast *operande_droit;
        creer_patchwork_binaire_fct creer_patchwork;
};

struct operation {
        enum arite_operation arite;
        union {
                struct operation_unaire  oper_un;
                struct operation_binaire oper_bin;
        } u;
};


struct valeur {
        enum nature_primitif nature;
        creer_patchwork_valeur_fct creer_patchwork;
};


struct noeud_ast_data {
        const char *nom;

        // nature du noeud: VALEUR ou OPERATION
        enum nature_noeud nature;
        // selon la nature, les donnees representant le noeud
        union {
                struct valeur val;       // si nature == VALEUR
                struct operation oper;   // si nature == OPERATION
        } u;
};
```

Annexe II : en-tête du module patchwork

```
#ifndef PATCHWORK_H
#define PATCHWORK_H

#include <stdint.h>

enum nature_primitif {
        CARRE,
        TRIANGLE,
        NB_NAT_PRIMITIFS     /* sentinelle */
};

enum orientation_primitif {
        EST,
        NORD,
        OUEST,
        SUD,
        NB_ORIENTATIONS /* sentinelle */
};

struct primitif {
        enum nature_primitif nature;
        enum orientation_primitif orientation;
};

struct patchwork {
        uint16_t hauteur, largeur;
        struct primitif **primitifs;    /* tableau de hauteur pointeurs
                                           vers des tableaux de largeur primitifs */
};

/* Cree et retourne un patchwork compose d'une image primitive,
 * de taille 1x1, de nature nat et d'orientation EST. */
extern struct patchwork *creer_primitif(const enum nature_primitif nat);

/* Cree et retourne un nouveau patchwork en appliquant a p une rotation
 * de 90 degres dans le sens direct. */
extern struct patchwork *creer_rotation(const struct patchwork *p);

/* Cree et retourne un nouveau patchwork par juxtaposition de p_g et
 * p_d (p_g a gauche de p_d).
 * Si les tailles ne sont par concordantes, retourne NULL. */
extern struct patchwork *creer_juxtaposition(const struct patchwork *p_g,
                                             const struct patchwork *p_d);

/* Cree et retourne un nouveau patchwork par superposition de p_h et
 * p_b (p_h au dessus de p_b).
 * Si les tailles ne sont par concordantes, retourne NULL. */
extern struct patchwork *creer_superposition(const struct patchwork *p_h,
                                             const struct patchwork *p_b);

/* Libere toute la memoire allouee pour le patchwork p. */
extern void liberer_patchwork(struct patchwork *p);

#endif /* PATCHWORK_H */
```

Annexe III : en-tête du module image

```
#ifndef IMAGE_H
#define IMAGE_H

#include "patchwork.h"

/* Cree une image du patchwork patch, a partir des deux images ppm
 * representant les images primitives carre et triangle.
 * Le resultat est enregistre dans ficher_sortie au format ppm P6.
 * Precondition: les deux images primitives sont carrees et de meme taille. */
extern void creer_image(const struct patchwork *patch,
                        const char *fichier_ppm_carre,
                        const char *fichier_ppm_triangle,
                        FILE *fichier_sortie);

#endif /* IMAGE_H */
```