TP 2
OUTIL MAKE

MOKRANI Abdeslam
LIN Groupe 1

Table des Matières

Introduction

L'objectif de ce TP est de réaliser un outil *make* minimal dont les commandes sont simplement affichées.

Les noms constituant les listes de cibles et les listes de dépendances ne correspondent pas à des noms de fichiers système existants, on doit donc simuler les différentes taches gérant les dates système de ces fichiers virtuels comme le fait le système d'exploitation. Pour cela, toutes les dates de ces fichiers sont mémorisées dans un fichier texte.
Si un fichier est créé alors une date doit lui être associée dans le fichier des dates, s'il est modifié, alors la date lui correspondant est mis à jour, finalement, s'il est supprimé, la ligne correspondant à sa date dans le fichier texte est supprimée.

Ces opérations sont effectuées soit par l'utilisateur en modifiant correctement le fichier texte des dates, soit par le programme lui-même.

1.Choix et étude d'une structure de données

Lors de l'exécution de l'outil *make*, les données du fichier *Makefile* doivent être représentées par un graphe orienté dans lequel :

- La valeur d'un sommet englobe la liste de noms des cibles et celle des commandes d'une règle. Chaque règle est donc représentée par un sommet dans le graphe.

- Les fils d'un sommet sont tous les sommets dont au moins une cible appartient à la liste des dépendances de la règle correspondant à ce sommet.

Exemple :
Soit le fichier makefile suivant :

```
a e : b c
      commande a 1
      commande a 2
      commande a 3
b : a d f h
      commande  b
d :
      commande  d 1
      commande  d 2
c : f g
      commande  c
f :
      commande f 1
      commande f 2
      commande f 3
      commande f 4
g :
      commande g 1
```

Voici le graphe correspondant à ce fichier :

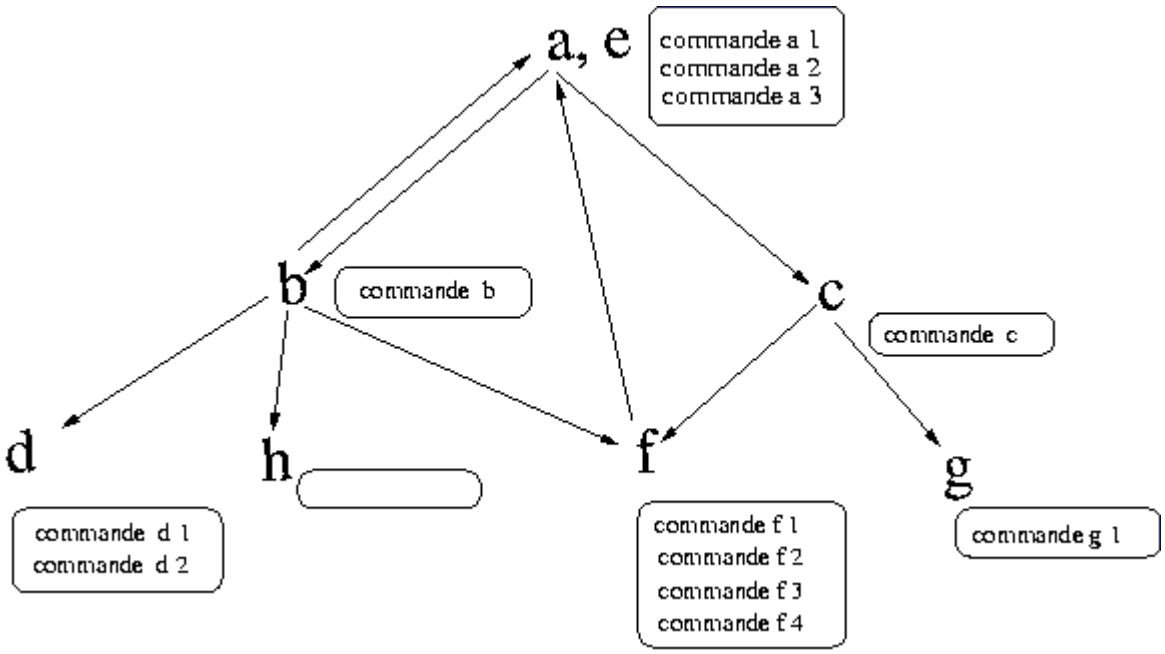


Figure 1

1.1.Représentation du graphe en mémoire

Au niveau mémoire, on choisit de représenter le graphe de dépendance par une liste d'adjacences, on associe alors à chaque valeur d'un sommet du graphe une liste d'éléments permettant de reconnaître tous ses fils, on pourra ainsi parcourir tout le graphe à partir d'un sommet donné. Dans notre cas, cette liste correspond à la liste de dépendances.
Pour pouvoir se déplacer d'un sommet à un autre, il faut réunir tous les sommets du graphe dans une seule liste.

Cette représentation a l'avantage de pouvoir minimiser le coût en mémoire grâce à l'utilisation de listes pouvant être implémentées dynamiquement, ce qui n'est pas le cas pour certaines autres représentations comme, par exemple, la représentation par matrice qui est implémentée par tableaux. Son point faible est le grand coût en temps d'exécution lié au parcours linéaire des listes contrairement à la représentation par matrice où l'accès aux éléments est direct.

On peut justifier notre choix de représentation du graphe par la structure du fichier *Makefile* qui ressemble à celle des listes d'adjacences, ce qui simplifie le code de la fonction de lecture du graphe et permet de diminuer le temps d'exécution en lecture du graphe.

1.2.Structure de données

On a choisit précédemment de représenter le graphe par liste d'adjacences, la structure de données la plus appropriée pour cela est les listes récursives. On utilise ainsi des listes récursives pour sauvegarder chaque liste de dépendances d'une règle (correspondant à un sommet), et une autre pour sauvegarder toutes les règles.

1.3.Implémentation

Pour mieux gérer la mémoire on va implémenter les listes récursives en utilisant des variables dynamiques. Plusieurs types de listes récursives sont possibles selon le type d'éléments qu'elles contiennent.

Pour les listes des noms, on crée le type TLiStrings de la façon suivante :

```
TLiStrings = ^TStrings; {type d'une liste récursive de chaîne de caractères}
TStrings   = record
               val   : string;
               suiv : TLiStrings;
             end;
```

On crée un type TRegle pour représenter les données d'une règle; ce type est un enregistrement dont les champs sont la liste des cibles, celle des commandes et celle des dépendances de la règle qui sont tous de type TLiStrings, ainsi qu'un booléen *marque* dont la valeur permet de savoir si la règle est déjà traitée pendant le parcours du graphe.
TRegle représente ainsi un sommet du graphe dont la valeur est la liste des cibles et celle des commandes. La liste des dépendances permet de déterminer quel sont les fils du sommet correspondant à la règle.
Voici le code qui implémente ce type :

```
TRegle   = record     {type d'une variable englobant les données d'une règle}
             listeCib,
             listeCom : TLiStrings;

             listeDep : TLiStrings;   {liste des fils d'un sommet dans le graphe}
             marque   : boolean;   {sert à détecter les cycles dans le graphe lors de
son                                          parcours}
           end;
```

Il reste à définir le type d'une liste récursive qui va contenir tous les variables de type TRegle du programme, voici le code correspondant à ce type :

```
TLiRegles  = ^TRegles; {type d'une liste récursive de variables de type TRegle}
TRegles    = record
               val   : TRegle;
               suiv   : TLiRegles; {sommet suivant dans le graphe}
```

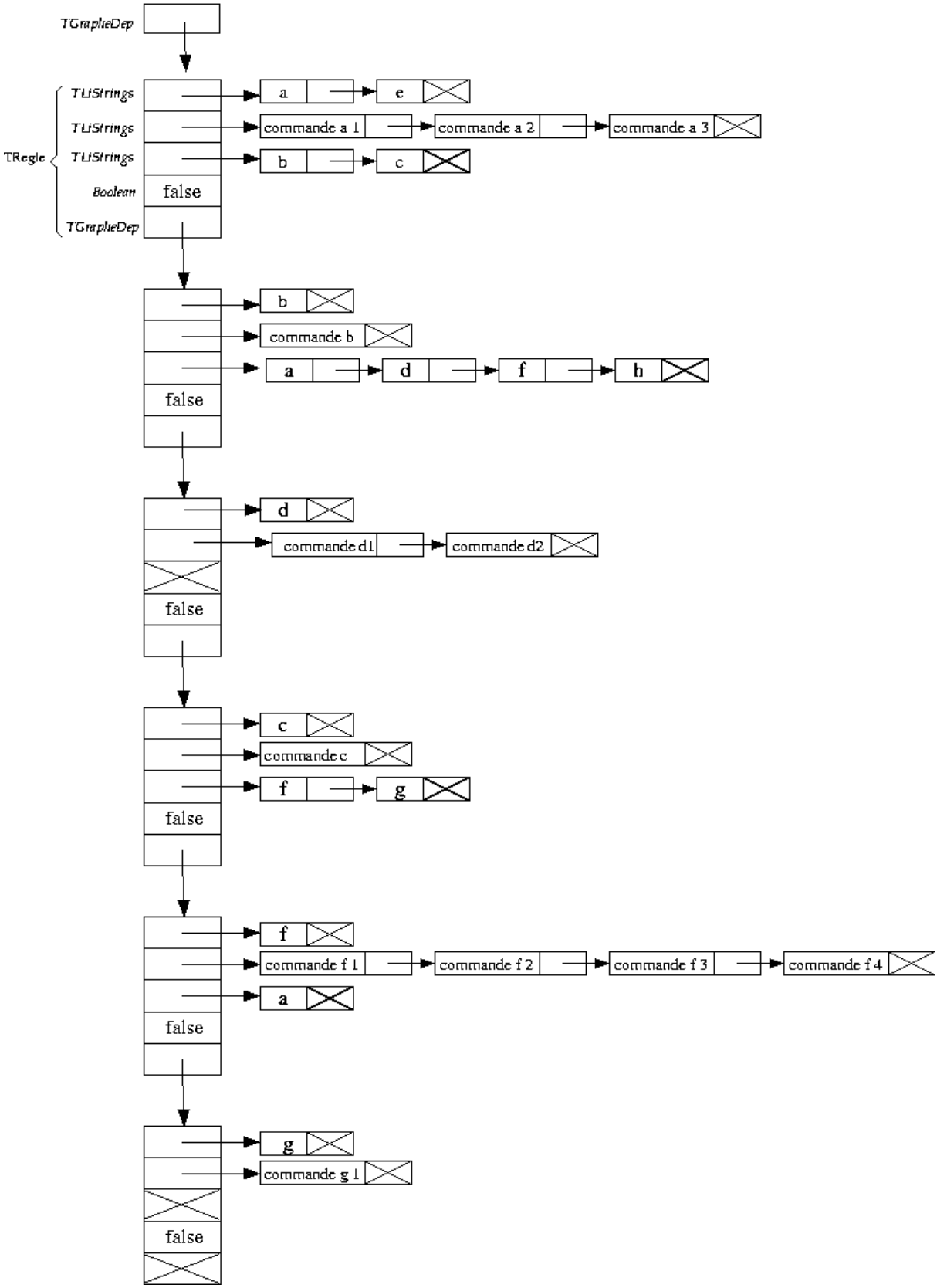
end;

Ce dernier type est donc aussi le type d'un graphe de dépendance :

TGrapheDep = TLiRegles; {définition de TGrapheDep}

On définit ensuite les opérateurs nécessaires pour gérer ces listes récursives, comme l'opérateur de construction d'un élément avec une liste, les opérateurs permettant d'accéder au premier élément et la suite d'une liste etc. Ces opérateurs sont surdéfinis pour les différents types de listes récursives.

Voici le schéma mémoire représentant le graphe de la *figure 1* :



2.Lecture d'un graphe de dépendance

On définit une fonction qui prend en paramètre le nom d'un fichier texte et retourne un graphe de dépendance correspondant à ce fichier. Si le fichier ne respecte pas la syntaxe d'un fichier *makefile***,** des erreurs indiquant l'emplacement dans le fichier et la cause de l'erreur sont déclenchées et le programme est arrêté.

Voici la description de l'algorithme de lecture :

début
- *on ouvre le fichier texte;*
- *s'il n'existe pas alors erreur;*
- *on crée le graphe qui va être retourné par la fonction;*
- *tant que la fin du fichier n'est pas atteinte faire*
- *début*
  - *on lit une ligne à partir du fichier, le pointeur du fichier passe à la ligne suivante;*
  - *tant que cette ligne est vide ou ne contient que des espaces et tabulations faire;*
    - *on lit la prochaine ligne du fichier;*
  - *si la ligne commence par une tabulation alors erreur (cette ligne correspond à une ligne de commande alors qu'aucune règle n'est encore lue);*
  - *on se place au début du premier mot de la ligne:*
  - *tant que la fin de la ligne n'est pas atteinte;*
  - *début*
    - *on lit le mot et on ce place au début du prochain mot;*
    - *si le mot correspond au caractère ':' alors*
    - *début*
      - *si aucune cible n'est lue dans la ligne alors erreur;*
      - *si le signe ':' était déjà lu alors erreur;*
    - *fin si*
    - *sinon*
      - *si le signe ':' est déjà lu alors on ajoute le mot à la liste des dépendances*
    - *sinon*
      - *on ajoute le mot à la liste des cibles;*
  - *fin tant que;*
  - *si le signe ':' n'est pas rencontré alors erreur;*
  - *on lit une ligne;*
  - *tant que la fin du fichier n'est atteinte la ligne lue correspond à une ligne de commande (elle commence par une tabulation) faire*
  - *début*
    - *on ajoute le reste de la ligne à la liste des commandes ;*
    - *on lit la ligne suivante;*
  - *fin tant que;*
  - *on ajoute la règle ainsi constituée au graphe;*
- *fin tant que;*
- *on retourne le graphe lu;*
- *on ferme le fichier;*

fin ;

3.Lecture des dates de fichiers

On crée un Type TDate qui permettra de stocker une date sous forme de valeurs numériques, un type TDateFich qui permet d'associer à la date un nom de fichier et un type TLiDatesFich qui représente une liste récursive dont les éléments sont de type TDateFich.

```
TDate      = record {type d'une variable représentant la date d'un fichier}
               j, m, a, h, min, sec : byte;
             end;
TDateFich  = record
               fich   : string; {le nom d'un fichier}
               date   : TDate;  {sa date de dernière modification}
             end;
TLiDatesFich = ^TDatesFich;
TDatesFich = record
               val  : TDateFich;
               suiv : TLiDatesFich;
             end;
```

On implémente les opérateurs nécessaires pour cette liste.

La fonction qui permet de lire les dates des fichiers à partir du fichier de dates prend en paramètre le nom du fichier à lire et retourne la liste récursive des dates de fichiers lues.
Voici l'algorithme en question :

```
début
    - si le fichier n'existe pas alors erreur
    - sinon on l'ouvre;
    - on crée la liste a retourner;
    - tant que la fin du fichier n'est pas atteinte
      début
        - on lit une ligne
        - si elle contient du texte alors
          début
            - on lit le premier mot de la ligne;
            - si le mot correspond au caractère ':' alors erreur;
            - on ajoute le mot à la liste des dates en tant que nom de fichier;
            - on lit le prochain mot;
            - ci ce mot n'est pas le caractère ':' alors erreur;
            - on lit le texte restant de la ligne;
            - s'il ne correspond pas à une date alors erreur;
            - on convertit le texte lu à une variable de type TDate et on l'ajoute à
              la liste en tant que date au fichier déjà lu;
          fin si;
      fin tant que;
    - on ferme le fichier;
fin;
```

4.Écriture des dates dans un fichier

La procédure qui écrit les dates dans un fichier prend en paramètre une variable de type TLiDatesFich qui contient les noms des fichiers associés à leurs dates, et le nom du fichier des dates dans lequel sera écrites les dates des fichiers.
Un nom de fichier associé à sa date lu dans la liste des dates est convertit à une chaîne de caractères selon la syntaxe d'une ligne du fichier des dates, elle est ensuite ajoutée au fichier des dates qui est ouvert en écriture. Ceci est répété pour chaque élément de la liste, on choisit donc d'implémenter cette procédure récursivement; toutefois l'ouverture et la fermeture du fichier des dates ne doit pas s'effectuer à chaque appel récursif, il faut donc utiliser deux procédures imbriquées où la première permet d'ouvrir le fichier puis appelle la deuxième pour l'écriture des dates et ferme le fichier, la deuxième procédure peut donc être implémentée récursivement.

Voici la description de l'algorithme correspondant à ces procédures :

procédure ecrireDates (prend en paramètre la liste et le fichier des dates);

procédure ecrireDatesAux (prend en paramètre la liste des dates de fichiers);
début
- si la liste des des dates de fichiers n'est pas vide alors
début
- on appelle la procédure crireDatesAux avec comme paramètre la suite de la liste courante;
- on convertit le nom du fichier et la date du premier élément de la liste en une chaîne de caractères puis on l'écrit dans le fichier des dates;
fin si;
fin ecrireDatesAux;

début
- on ouvre le fichier en écriture;
- on appelle la procédure ecrireDatesAux avec comme paramètre la liste des dates;
- on ferme le fichier;
fin ecrireDates;

5.Parcours de graphe et détection de cycles

La fonction du parcours du graphe de dépendance prend en paramètre le graphe à parcourir, une chaîne de caractères représentant la cible à générer et le nom du fichier contenant les dates des différentes cibles.
Si au moins un fichier de dépendance est plus récent que la cible, alors les commandes lui correspondant doivent être exécutées pour pouvoir tenir compte des modifications survenues sur les dépendances. Mais avant cela, il faut d'abord parcourir le graphe à partir de chaque dépendance modifiée et différente de la cible, si elle existe en tant que cible, afin d'effectuer les modifications qu'elles entraînent sur le graphe. Cela montre que ce parcours est en profondeur.
Par contre, si toutes les dépendances sont plus anciennes que la cible, alors celle-ci est déjà mise à jour donc pas de parcours à faire.

Le parcours du graphe est toujours fini car on ne parcourt jamais le graphe à partir d'une cible déjà traitée et le nombre de cibles est fini. Dans le programme, c'est la variable booléenne *marque* qui indique si la cible est traitée.

On définit les deux fonctions suivantes :

- *getDateFich (li : TLiDatesFich; nomFich : string) : Tdate;*
  qui recherche une date du fichier dont le nom est nomFich dans la liste puis retourne la date si elle est trouvée, sinon elle retourne une date dont le jour est égale à 0 pour montrer que le fichier n'existe pas et qu'il doit être crée;

- *modifDateFich (li TLiDatesFich; nomFich : string) : TLiDatesFich;*
  qui permet de remplacer la date du fichier nomFich avec la date actuelle du système dans la liste li si nomFich est trouvé, sinon le nom du fichier et la date système sont ajoutés à la liste comme un nouvel élément; la liste modifiée est ensuite retournée.

On choisit d'utiliser dans l'algorithme de parcours une procédure récursive pour le parcours est l'affichage des commandes, il faut donc une autre procédure pour l'ouverture du fichier des dates, sa lecture et l'écriture des dates dans ce fichier.

L'algorithme du parcours se résume à deux procédures et une fonction dont les descriptions suivent :

procédure parcourir(prend en paramètres le graphe à parcourir, la cible et le fichier des dates)
elle a comme variable locale une liste de type TLiDatesFich qui va contenir toutes les dates des fichier lues à partire du fichier des dates.

procédure parcourireAux(prend en paramètres le graphe à parcourir et la cible);

fonction actualiser(prend en paramètre une liste de dépendance et la date de la cible, elle renvoie true si des actualisations sont effectuées, false sinon);
début
- on initialise la valeur du retour à false;
- si la liste n'est pas vide alors
début
- on actualise la suite de cette liste, la valeur du retour prend true si des actualisations sont effectuées ;
- on calcule la date du fichier dont le nom est celui du premier élément de la liste, cela grâce à la fonction GetDateFich;
- si cette date correspond à une date de fichier qui n'existait pas ou elle est plus récente que la date passée en paramètre, alors
début
- la valeur du retour prend true;
- on parcourt le graphe à partir de la cible correspondant au premier élément de la liste ;
fin si;
fin si;
fin;

début
- si le graphe n'est pas vide alors
début
- si la cible appartient à la liste des cibles du premier élément de la liste représentant le graphe alors
début
- si la règle lui correspondant n'est pas marquée (marque<>false) alors
début
- on marque cette règle (marque := true);
- si la liste de dépendances n'est pas vide alors
début
- on actualise cette liste par rapport à la date de la cible;
- si l'actualisation rend true alors
début
- on affiche les commandes de la règle car elles doivent être exécutées;
- on actualise la date de la cible dans la liste des dates;
fin si;
fin si

```
        sinon (la liste de dépendances est vide)
            - on affiche les commandes de la règle;
            - on actualise la date de la cible dans la liste des dates;
        fin si;
    fin si;
fin si
sinon (la cible n'est pas dans le premier élément de la liste graphe)
    - on parcourt le sous graphe correspondant à la suite de la liste
      graphe à partir de la cible;
fin si; (si le graphe est vide alors pas de parcours)
fin;

Début
    - on lit les dates des fichiers dans la liste des dates à partir du fichier de dates;
    - on appelle la fonction récursive de parcours avec comme paramètres le
      graphe et la cible;
    - on enlève le marquage de toutes les règle du graphe;
    - on écrit les dates de fichiers contenues dans la liste des dates ainsi modifiée
      dans le fichier des dates;
    - on libère la mémoire occupée par la liste des dates;
fin;
```