Autres outils de la chaine de production de programmes

A. Le préprocesseur

Le préprocesseur est un outil exécuté en amont de la compilation. Il effectue des modifications textuelles sur le fichier source à partir de *directives*. Les différentes directives au préprocesseur, introduites par le caractère `#`, permettent :

- l'incorporation de fichiers source (`#include`),
- la définition de constantes symboliques et de macros (`#define`),
- la compilation conditionnelle (`#if`, `#ifdef`,...).

1. La directive `#include`

Cette directive permet d'inclure le contenu d'un autre fichier dans le fichier source. Ce dernier peut être un fichier en-tête de la librairie standard (`stdio.h`, `math.h`,...) ou n'importe quel autre fichier. La directive `#include` possède deux syntaxes voisines :

```
#include <nom-de-fichier>   recherche le fichier mentionné dans un ou
plusieurs répertoires systèmes définis par l'implémentation (par exemple, /
usr/include/) ;

#include "nom-de-fichier" recherche le fichier dans le répertoire courant
(celui où se trouve le fichier source).
```

La première syntaxe est généralement utilisée pour les fichiers en-tête de la librairie standard, tandis que la seconde est plutôt destinée aux fichiers créés par l'utilisateur.

```
Exemples
#include <stdio.h>
#include "mes_définitions.h"
```

2. La directive `#define`

La directive `#define` permet de définir des constantes symboliques.
La directive `#define` *`nom définition`* demande au préprocesseur de substituer toute occurence de *`nom`* par la chaîne de caractères *`définition`* dans la suite du fichier source.

Exemple :

```
#define TAILLE_CHAINE 15
```

3. La compilation conditionnelle

La *compilation conditionnelle* a pour but d'incorporer ou d'exclure des parties du code source dans le texte qui sera généré par le préprocesseur. Elle permet d'adapter le programme au matériel ou à l'environnement sur lequel il s'exécute, ou d'introduire dans le programme des instructions de débogage.
La syntaxe de la directive est

```
#ifdef symbole
  Morceau1_code
#else
  Morceau2_code
#endif
```

Si *symbole* est défini au moment où l'on rencontre la directive `#ifdef`, alors *Morceau1_code* sera compilé et *Morceau2_code* sera ignoré. Dans le cas contraire, c'est *Morceau2_code* qui sera compilé. La directive `#else` est facultative.

```
#define TOTO
  ....
#ifdef TOTO
    printf("%s\n",”c’est le morceau 1”);
#else
    printf("%s\n",”c’est le morceau 2”);
#endif /* DEBUG */
```

4. Un exemple

Soit le code suivant :

```
/* fichier 1 mes_variables.h */
int haut ;
int pile[1..taille_pile] ;
```

```
/* fichier gestion_pile */

# define taille_pile 20
#define message_erreur " debordement de
pile "
# include " mes_variables.h "

procedure EMPILER(x)
debut
        Pile(haut):=x;
        Si (haut < taille_pile) haut := haut + 1
        Sinon  afficher (message_erreur) ; fsi
fin
procedure DEPILER(x)
debut
        haut :=haut - 1;
        x := Pile(haut);
fin
```

La compilation du module gestion_pile fait d’abord appel au préprocesseur qui en exécutant les directives transforme le fichier comme suit :

/* fichier gestion_pile */

```
int haut ;
```
int pile[1..taille_pile] ; ← # include " mes_variables.h "

procedure EMPILER(x)
debut
Pile(haut):=x;
Si (haut < 20) haut := haut + 1 ← define taille_pile 20
Sinon afficher (debordement de pile) ; fsi ← #define message_erreur " debordement de pile "

fin
procedure DEPILER(x)
debut
haut :=haut - 1;
x := Pile(haut);
fin

B. Le Make

Le Make est un outil qui exploite les dépendances existantes entre les modules entrant en jeu dans la construction d'un programme exécutable pour ne lancer que les opérations de compilations et éditions de liens nécessaires, lorsque ce programme exécutable doit être reconstruit suite à une modification intervenue dans les modules sources.

Exemple

Dans le programme suivant x.c, l'inclusion du fichier defs par l'ordre #include "defs" crée une dépendance entre ces deux modules.

```
/* fichier x.c */
#include "defs"
main()
{
    ...
}
```

Cet outil utilise deux sources d'informations : un fichier de description appelé le Makefile qui contient la description des dépendances entre les modules et les noms et les dates de dernières modifications des modules.

1. Format du fichier Makefile

Le fichier Makefile décrit les dépendances existantes entre les modules intervenant dans la construction d'un exécutable : il traduit sous forme de règles le graphe de dépendance du programme exécutable à construire et indique pour chacune de ces dépendances, l'action qui lui est associée.

Une règle dans le fichier Makefile est de la forme :

module cible : dépendances

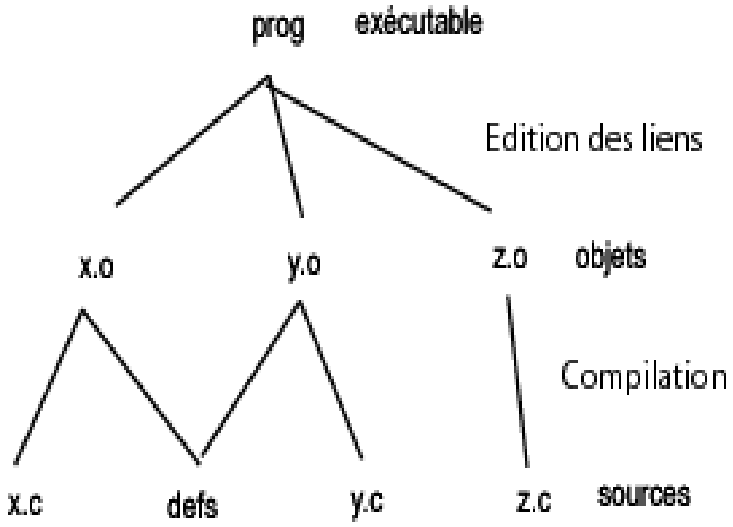
commande pour construire le module cible

Prenons comme exemple le cas suivant : le programme exécutable prog est construit à partir d'une étape d'édition des liens prenant en compte les trois modules objets x.o, y.o et z.o. Le module objet z.o est issu de la compilation d'un programme source z.c. Les modules x.o et y.o sont à leur tour issus de la compilation respective des modules source x.c et y.c. Ces deux derniers modules utilisent un module defs par le biais d'un ordre d'inclusion #include. Le graphe de dépendance du programme prog est donné sur la figure 3.11.

Le fichier Makefile résultant est :

```
prog : x.o y.o z.o
ld x.o y.o z.o - o prog
x.o : defs x.c
cc - o x.c
y.o : defs y.c
cc - c y.c
z.o : z.c
cc - c z.c
```

La première règle stipule la dépendance associée au programme exécutable prog qui est donc construit à partir des modules x.o, y.o et z.o. La commande permettant la construction du programme exécutable prog à partir de ces trois modules objets est la commande d'édition des liens ld x.o y.o z.o - o prog.



La seconde et la troisième règle stipulent que le fichier objet x.o (respectivement y.o) dépend à la fois du fichier x.c (respectivement y.c) et du fichier defs. Les fichiers y.o ou x.o sont construits par compilation des fichiers.c correspondants.

Enfin, la dernière règle spécifie que le fichier z.o dépend uniquement de la compilation du fichier z.c.

2. Fonctionnement de l'utilitaire Make

L'outil Make utilise le fichier Makefile et les dates de dernières modifications des modules pour déterminer si un module est à jour. Un module est à jour si : le module existe et si sa date de dernière modification est plus récente que les dates de dernière modification de tous les modules dont il dépend ou bien elle est égale.

Si un module n'est pas à jour, la commande associée à ses dépendances est exécutée pour reconstruire le module.

Considérons par exemple que le module z.c soit modifié. L'utilitaire Make va détecter que ce module est devenu plus récent que le module objet z.o. Il va donc lancer la commande associée à la règle de dépendance du module z.o, soit la commande de compilation cc -c z.c. L'exécution de cette commande va à son tour générer un module z.o plus récent que le programme exécutable prog. En conséquence, l'utilitaire Make va reconstruire le programme exécutable prog en lançant la commande d'édition des liens ld x.o y.o z.o -o prog. D'une façon similaire, toute modification au sein du module defs entraînera la reconstruction des modules y.o et x.o, par le biais de deux opérations de compilation, puis la reconstruction du programme exécutable prog. Dans ces deux cas, seules les opérations de compilation ou d'édition des liens nécessaires sont exécutées.

L'utilitaire Make est appelé au moyen de la commande make prog qui suppose qu'un fichier Makefile est présent dans le répertoire où la commande est lancée. La commande make -f nom_fichier prog fait également appel à l'utilitaire Make mais avec un fichier de dépendance appelé nom_fichier et non Makefile.