Université de Nice-Sophia Antipolis — Licence d'Informatique
Module d'analyses — 2004–2005

Examen de contrôle continu

Durée : 2 heures
Tous documents autorisés

Analyse d'un fichier `Makefile`

Le programme `make` détermine automatiquement quelles composantes d'un projet nécessitent d'être regénérées, et exécute les commandes permettant cette génération. Ce programme utilise un fichier `Makefile` pour déduire quelles composantes du projet doivent être mises à jour et comment.

Ce fichier `Makefile` contient une série de *règles* de la forme

```
cible1 cible2... : prérequis1 prérequis2...
        commande1
        commande2
        ...
```

où les *prérequis* doivent être à jour pour obtenir les *cibles* en utilisant les *commandes*.

Le but de cet examen de contrôle continu va être d'analyser les règles d'un fichier `Makefile`.

1 Exemple de fichier

Voici le fichier `Makefile` utilisé dans la correction de cet examen :

```
objects = mklex.o mkparse.o mkmain.o
CFLAGS = -Wall -pedantic
LEX = flex

mk: ${objects}
        ${CC} ${objects} -o $@

${objects}: mkdefs.h

.PHONY: clean test check nothing
nothing:
test check: mk Makefile; grep -v \= Makefile | ./mk
clean:
        rm -f ${objects}\
              mk\
              mklex.c\
              *~
```

Les trois premières lignes qui contiennent des affectations seront ignorées.

2 Analyse lexicale

L'analyse de ce fichier pose problème car les tabulations et les retours à la ligne ont une valeur syntaxique dans certaines circonstances et sont de simples blancs dans d'autres. Gérer tous ces

cas au moment de l'analyse syntaxique serait très compliqué, alors que l'analyse lexicale peut s'en charger assez simplement.

1. Un mot est une suite non vide de caractères qui
   - soit ne sont pas :
     - des blancs (espace, tabulation ou retour à la ligne)
     - des caractères réservés (les deux points ou le point-virgule qui servent de séparateurs)
     - une contre-oblique `\`,
   - soit sont une contre-oblique `\` suivie de n'importe quel caractère, à l'exception du retour à la ligne.
2. En effet, les retours à la lignes précédés d'une contre-oblique `\` sont eux interprétés comme de simples espaces et ignorés lors de l'analyse syntaxique.
3. Les commandes sont séparées,
   - soit par un retour à la ligne suivi d'une tabulation,
   - soit par un point-virgule.
4. Les règles sont séparées par au moins un retour à la ligne.

Soient les déclarations C suivantes dans le fichier d'en-têtes :

```
/* définitions des lexèmes particuliers */
#define END   0
#define MOT 256
#define SR  257 /* un séparateur de règles */
#define SC  258 /* un séparateur de commandes */
extern char *mot;
```

a. Écrivez les règles FLEX qui renvoient les bonnes valeurs de lexèmes (`END`, `MOT`, `SR`, `SC`, et le caractère ':') rencontrés. Dans le cas où ce lexème est un mot, mettez la chaîne reconnue dans la variable `mot`.

```
mot               ([^ \\\t\n:;]|\\[^\n])+

%%
{mot}             mot = yytext; return MOT;
\n\t|;            return SC;
\\\n              ;
\n+               return SR;
:                 return yytext[0];
[ \t]             ;
.                 { fprintf (stderr, "Lexeme␣inconnu␣ligne␣%d␣:␣%c\n",
                             yylineno, yytext[0]); }
```

.....

3 Analyse syntaxique

Voici maintenant un exemple de grammaire pour les règles d'un `Makefile` dans une syntaxe abstraite :

Grammaire G_1

Axiome = Makefile
N_1 = {Makefile, Règle, Cibles, Prérequis, Commandes}
T_1 = {MOT, SR, SC, :}
P_1 = { Makefile $\rightarrow_1$ (Règle SR)$^+$ }
Règle $\rightarrow_2$ Cibles : Prérequis ? Commandes ?
Cibles $\rightarrow_3$ MOT$^+$
Prérequis $\rightarrow_4$ MOT$^+$
Commandes $\rightarrow_5$ (SC MOT$^+$)$^+$

b. Proposez une grammaire algébrique concrète LL(1) équivalente à la grammaire G_1. Numérotez ses productions en vue des exercices suivants.

Grammaire G_2

Axiome = Makefile
N_1 = {Makefile, Restemk, Règle, Mots, Restemot, Prérequis, Commandes}
T_1 = {MOT, SR, SC, :}
P_1 = { Makefile $\rightarrow_1$ Règle SR Restemk }
Restemk $\rightarrow_2$ Makefile
Restemk $\rightarrow_3 \nu$
Règle $\rightarrow_4$ Mots : Prérequis Commandes
Mots $\rightarrow_5$ MOT Restemot
Restemot $\rightarrow_6$ Mots
Restemot $\rightarrow_7 \nu$
Prérequis $\rightarrow_8$ Mots
Prérequis $\rightarrow_9 \nu$
Commandes $\rightarrow_{10}$ SC Mots Commandes
Commandes $\rightarrow_{11} \nu$

. . . . . .

c. Calculez les premiers et suivants pour cette grammaire.

	PREM$_1$	SUIV$_1$
Makefile	MOT	$\vdash$
Restemk	MOT, $\vdash$	$\vdash$
Règle	MOT	SR
Mots	MOT	SR, SC, :
Restemot	MOT, $\vdash$	SR, SC, :
Prérequis	MOT, $\vdash$	SR, SC
Commandes	SC, $\vdash$	SR

. . . . . .

d. Donnez la table d'analyse LL(1) pour cette grammaire. Pour rappel, cette table contient le numéro i de la production $A \rightarrow_i \alpha$ à l'intersection de la ligne du non terminal A avec la colonne du terminal a si et seulement si $a \in \text{PREM}_1(\alpha\ \text{SUIV}_1(A))$.

	:	MOT	SR	SC	$\vdash$
Makefile		1			
Restemk		2			3
Règle		4			
Mots		5			
Restemot	7	6	7	7	
Prérequis		8	9	9	
Commandes			11	10	

.....

e. Donnez le code C des procédures **void** `Prerequis(`**void**`)` et **void** `Commandes(`**void**`)` lors d'une analyse récursive descendante de votre grammaire. On supposera pour cela

- que la variable `lexeme` de type **int** contient le premier lexème non consommé de l'entrée au moment de l'appel de la procédure,
- que l'on dispose de la fonction **int** `lex (`**void**`)` qui consomme un terminal et retourne la valeur du premier terminal non encore consommé,
- que les procédures reconnaissant les autres non terminaux sont disponibles,
- et que l'on dispose d'une procédure de report d'erreurs.

```
void
Prerequis (void)
{
  switch (lexeme)
    {
    case MOT:
      AFFTRACE ("Prerequis -> Mots\n");
      Mots ();
      break;
    case SC: case SR:
      AFFTRACE ("Prerequis -> vide\n");
      break;
    default:
      erreur ("Prerequis", "MOT ou SC ou SR");
    }
}

void
Commandes (void)
{
  switch (lexeme)
    {
    case SC:
      AFFTRACE ("Commandes -> SC Mots Commandes\n");
      lexeme = yylex ();
      Mots ();
      Commandes ();
      break;
    case SR:
      AFFTRACE ("Commandes -> vide\n");
      break;
    default:
      erreur ("Commandes", "SC ou SR");
    }
}
```

.....