UNIVERSITÉ BORDEAUX I — U.F.R. Mathématiques et Informatique
Session de Janvier 1997 — J.Bétréma

Maîtrise d'Informatique
Compilation (tronc commun)
(Durée: 3 heures)

Tous documents interdits.

1 Analyse lexicale

On considère un texte (probablement produit par un tableur), dont chaque ligne représente un *enregistrement*. Un enregistrement est constitué d'un nombre variable de *champs*, et le séparateur de champs est le caractère de tabulation; un champ peut être vide. On distingue trois types de champs : entier, positif ou négatif; réel, positif ou négatif (on ne se préoccupera pas des cas particuliers tels que .5 ou 3.) ; tout autre champ est traité comme une chaîne de caractères.

On suppose disponibles :

- un type `Pointeur`;
- des fonctions d'allocation mémoire :

```
Pointeur allouer_chaine (char *s);
Pointeur allouer_entier (int n);
Pointeur allouer_reel (double x);
```

- des constantes `CMAX, EMAX`;
- la déclaration :

```
typedef Pointeur Enregistrement [CMAX];
```

Ecrire un analyseur lexical, dans le format reconnu par *lex*, qui lit un texte du type décrit ci-dessus, et remplit un tableau d'au plus `EMAX` enregistrements. La fonction produite retourne le nombre d'enregistrements lus, et a pour prototype :

```
int tableur (Enregistrement *t);
```

La réponse doit être simple mais précise; une réponse d'une complexité excessive vous fera perdre du temps sans vous rapporter aucun point. Commenter les (quelques) points délicats de programmation *lex*.

2 Analyse syntaxique

Soit G la grammaire ETF, à laquelle on a ajouté une règle d'affectation, où, comme en C, une affectation peut être considérée à son tour comme une expression. Voici le fichier, dans le format reconnu par *yacc*, qui décrit cette grammaire :

```
%token ID

%%
expr    : ID '=' expr
        | expr '+' terme
        | terme
        ;
terme   : terme '*' facteur
        | facteur
        ;
facteur : '(' expr ')'
        | ID
        ;
```

et en annexe, le fichier produit par l'option `-v` de yacc (en fait, `bison`). **Attention :** la lecture de cette annexe est indispensable à la résolution des questions de cette partie.

1. G est-elle ambiguë ? On demande une réponse argumentée brièvement, mais précisément.

2. G est-elle SLR(1) ? On demande une réponse argumentée brièvement, mais précisément.

3. Détailler le fonctionnement de l'analyseur syntaxique pendant l'analyse de

$$a = b + c$$

en utilisant une table à trois colonnes : action (décalage ou réduction), état de l'automate, contenu de la pile. Utiliser évidemment les numéros d'états et de règles fournis en annexe.

4. Pourquoi *yacc* ne signale-t-il pas de conflit dans l'état 1 ?

5. Expliquer le conflit de l'état 10, et sa résolution par yacc. Expliquer la raison du choix de yacc ; ce choix est-il le bon ?

6. Comment transformer la grammaire G pour éviter le conflit de l'état 10 ? Soit G' la nouvelle grammaire ; si cette grammaire est compilée avec l'option `-v`, quelles seront les différences entre le fichier produit et celui de l'annexe ?

3 Attributs

Soit G la grammaire des listes suivante :

$$S \to (L) \mid id$$
$$L \to L, S \mid S$$

Soit par exemple la liste $\lambda = (a, (b, c), a)$; a (de même que b ou c) est un *atome* de λ, et (b, c) (de même que a) un *composant* de λ. Ecrire un programme, dans le format reconnu par *yacc*, qui calcule (et affiche) simultanément :

1. le nombre d'atomes (distincts ou non) d'une liste (dans le cas de λ, le résultat est 4) ;

2. le nombre de composants (distincts ou non) d'une liste (dans le cas de λ, le résultat est 3) ;

3. la profondeur d'emboîtement d'une liste (dans le cas de λ, le résultat est 2).

4 Annexe

```
State 10 contains 1 shift/reduce conflict.

Grammar
rule 1    expr -> ID '=' expr
rule 2    expr -> expr '+' terme
rule 3    expr -> terme
rule 4    terme -> terme '*' facteur
rule 5    terme -> facteur
rule 6    facteur -> '(' expr ')'
rule 7    facteur -> ID

state 0

    ID   shift, and go to state 1
    '('  shift, and go to state 2

    expr go to state 3
    terme go to state 4
    facteur go to state 5

state 1

    expr  ->  ID . '=' expr   (rule 1)
    facteur  ->  ID .   (rule 7)
```

```
    '='  shift, and go to state 6

    $default reduce using rule 7 (facteur)

state 2

    facteur  ->  '(' . expr ')'   (rule 6)

    ID   shift, and go to state 1
    '('  shift, and go to state 2

    expr go to state 7
    terme go to state 4
    facteur go to state 5

state 3

    expr  ->  expr . '+' terme   (rule 2)

    $    go to state 15
    '+'  shift, and go to state 8

state 4

    expr  ->  terme .   (rule 3)
    terme  ->  terme . '*' facteur   (rule 4)

    '*'  shift, and go to state 9

    $default reduce using rule 3 (expr)

state 5

    terme  ->  facteur .   (rule 5)

    $default reduce using rule 5 (terme)

state 6

    expr  ->  ID '=' . expr   (rule 1)

    ID   shift, and go to state 1
    '('  shift, and go to state 2

    expr go to state 10
    terme go to state 4
    facteur go to state 5
```

```
state 7

    expr  ->  expr . '+' terme   (rule 2)
    facteur  ->  '(' expr . ')'   (rule 6)

    '+'  shift, and go to state 8
    ')'  shift, and go to state 11

state 8

    expr  ->  expr '+' . terme   (rule 2)

    ID   shift, and go to state 12
    '('  shift, and go to state 2

    terme go to state 13
    facteur go to state 5

state 9

    terme  ->  terme '*' . facteur   (rule 4)

    ID   shift, and go to state 12
    '('  shift, and go to state 2

    facteur go to state 14

state 10

    expr  ->  ID '=' expr .   (rule 1)
    expr  ->  expr . '+' terme   (rule 2)

    '+'  shift, and go to state 8

    '+'  [reduce using rule 1 (expr)]
    $default reduce using rule 1 (expr)

state 11

    facteur  ->  '(' expr ')' .   (rule 6)

    $default reduce using rule 6 (facteur)

state 12

    facteur  ->  ID .   (rule 7)

    $default reduce using rule 7 (facteur)
```

```
state 13

    expr  ->  expr '+' terme .   (rule 2)
    terme  ->  terme . '*' facteur   (rule 4)

    '*'  shift, and go to state 9

    $default reduce using rule 2 (expr)

state 14

    terme  ->  terme '*' facteur .   (rule 4)

    $default reduce using rule 4 (terme)

state 15

    $    go to state 16

state 16

    $default accept
```