Faculté des Sciences
Aix*Marseille Université

Année universitaire 2015/2016

Site : ☒ Luminy ☒ St-Charles ☐ St-Jérôme ☐ Cht-Gombert ☒ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☒ 2ème semestre ☐ Session 2 Durée de l'épreuve : 2h
Examen de : L3 Nom du diplôme : Licence d'Informatique
Code du module : ENSIN6U1 Libellé du module : Compilation
Calculatrices autorisées : NON Documents autorisés : NON

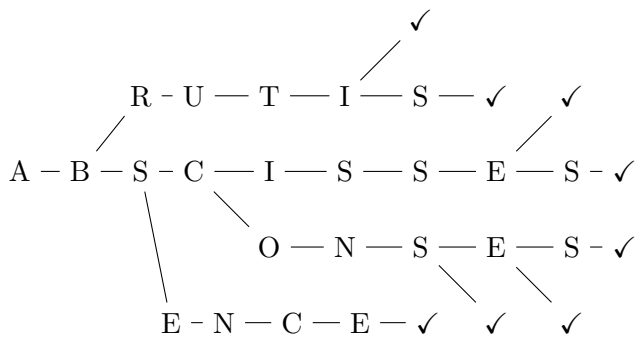
---

1 Compression de dictionnaires (14pt)

Un *arbre de préfixes* (AP) permet de compresser un *dictionnaire* (D) contenant une liste de mots. Le principe de compression d'un AP est de factoriser tous les préfixes communs à plusieurs mots. La liste de mots de D correspond à tous les chemins dans l'AP menant de la racine à une feuille.
Chaque noeud d'un AP est un caractère. Un mot de D est obtenu en parcourant l'AP à partir de la racine et en concaténant les caractères des noeuds jusqu'à une feuille. Les feuilles marquent la fin d'un mot avec ✓.
La figure (a) ci-dessous montre un exemple de dictionnaire D. Ce dictionnaire peut être compressé par un AP (figure b), représenté au format textuel compact (figure c).

```
abruti
abrutis
abscisse
abscisses
abscons
absconse
absconses
absence
absence
absences
absent
```

(a) Dictionnaire D.



(b) Abre de préfixes AP correspondant a D.

```
ab(ruti(,s),s(c(isse(,
s),ons(,e(,s))),ence))
```

(c) Abre de préfixes AP au format textuel compact.

Nous définissions le format textuel compact de l'exemple (c) comme suit :
- Un noeud de l'AP est représenté par un caractère quelconque `c`.
- Si un noeud contient un seul fils, ce fils peut être concaténé immédiatement à droite de son père.
- Si un noeud contient plus d'un fils, la liste des fils est donnée entre parenthèses, séparée par des virgules.

La grammaire G_{AP} ci-dessous permet de reconnaître un AP au format textuel compact. Les noeuds non-terminaux A et F sont indicés dans les parties droites. Le terminal `c` représente un caractère quelconque :

1. $A \rightarrow$ `c` A_1
2. $A \rightarrow$ `(` A_1 `,` A_2 F `)`
3. $A \rightarrow \varepsilon$
4. $F \rightarrow$ `,` A F_1
5. $F \rightarrow \varepsilon$

Exercice 1 Dessinez l'arbre de dérivation de l'AP `ce(,s)` qui compresse le dictionnaire D={`ce`, `ces`}.

Exercice 2 Un étudiant de L2 suggère d'ajouter la règle $A \rightarrow$ `c` à la grammaire G_{AP}. Expliquez en 3-4 phrases pourquoi c'est une mauvaise idée, si on veut écrire un analyseur $LL(1)$.

Exercice 3 Un étudiant de L2 suggère de remplacer la règle 5 de la grammaire G_{AP} par la règle 5′. $F \rightarrow$ `,` A. Expliquez en 3-4 phrases pourquoi c'est une mauvaise idée, si on veut écrire un analyseur $LL(1)$.

Exercice 4 Écrivez des grammaires attribuées qui permettent de calculer les informations listées ci-dessous sous la forme d'attributs pour un AP quelconque. Indiquez si les attributs sont synthétisés ou hérités. Pour chaque question, vous pourrez utiliser les attributs que vous avez défini pour répondre aux questions précédentes.

1. Nombre total de caractères n dans l'AP, y compris virgules et parenthèses.
2. Nombre m de mots contenus dans le dictionnaire D représenté par l'AP.
3. Longueur l du mot commençant à la racine de l'AP et terminant dans le noeud courant.
4. Longueur l_{max} du mot le plus long représenté dans l'AP.
5. Nombre total de caractères d dans le dictionnaire D représenté par l'AP, y compris le séparateur `'\n'`.
6. Taux de compression $c = n/d$.

Exercice 5 Écrivez une grammaire attribuée qui affiche la liste de mots de D lors de l'analyse d'un AP avec G_{AP}. En plus des attributs, vous pouvez utiliser l'action spéciale `print` pour afficher un mot.

Exercice 6 Calculez les ensembles PREMIER et SUIVANT des noeuds non-terminaux de la grammaire G_{AP}.

Exercice 7 Écrivez un compilateur en C qui permet de décompresser un AP au format textuel compact. Il doit afficher à l'écran le dictionnaire D décompressé correspondant, un mot par ligne. Vous pouvez utiliser les fonctions fournies ci-dessous.

```c
char uc, yyval;
FILE *yyin;
int yylex(){
  uc = fgetc(yyin);
  if( uc == '(' || uc == ')' || uc == ',' || uc == EOF ){ return uc; }
  else if(uc >= 'a' && uc <= 'z'){ yyval = uc;  return 'c'; /* 'c' -> caractere quelconque */}
  else{ printf( "caractere invalide %c\n", uc );  exit(-1); }
}
...
int main(int argc, char *argv[] ){
    yyin = fopen( argv[1], "r" );
    ... // Votre code ici
}
```

2 Assembleur (8pt)

Soit le programme L suivant :

```
1  entier $i, entier $a, entier $b;
2  main()    {
3    $a=2;    $b=8;    $i=1;
4    tantque ( 0 < $b ) faire {
5      $i=$i*$a;
6      $b=$b-1;
7    }
8    ecrire($i);
9  }
```

Exercice 1 Que fait ce programme ? Quelle est la valeur affichée à la fin de l'exécution ?

Exercice 2 Dessinez l'arbre abstrait associé à la fonction `main`.

Exercice 3 Ecrivez le programme MIPS correspondant à la ligne 4. Vous pouvez faire une gestion libre des registres, vous n'êtes pas obligé de passer systematiquement par la pile comme en TP.

Voici une autre version du programme précédent, utilisant une vision récursive du problème :

```
1  test( entier $a , entier $b ) {
2    si ( $b=0 ) alors {
3      retour 1;
4    }
5    si ( $b=1 ) alors {
6      retour $a;
7    }
8    si (($b%2)=0) alors {
9      retour test($a*$a,$b/2);
10   }
11   sinon {
12     retour $a*test($a,$b-1);
13   }
14 }
15
16 main() { ecrire(test(2,8)); }
```

Exercice 4 Cette version utilise l'opérateur modulo (%), de même précédence que la multiplication et la division. Qu'est-ce que l'introduction de cet opérateur va modifier dans la grammaire de L ?

Exercice 5 Ecrivez la fonction C pour la génération du code MIPS de cet opérateur modulo dans la fonction de parcours de l'arbre abstrait.

Exercice 6 Expliquez en quoi cette nouvelle version est optimisée en termes de complexité par rapport à la première version itérative. Écrivez en L une version itérative de ce nouvel algorithme.

Annexe : liste partielle d'opérations MIPS

`lw`	reg, adresse	charge la valeur pointée par l'adresse (4 octets)
`sw`	reg, adresse	sauvegarde le registre à l'adresse
`li`	reg, valeur	met la valeur dans le registre
`move`	reg, reg	copie le registre de droite vers celui de gauche
`not`	résultat, arg	met dans résultat la négation de arg
op	résultat, arg1, arg2	où *op* $\in$ {`add, sub, and, or, xor`}
opi	résultat, arg1, valeur	où *opi* $\in$ {`addi, andi, ori, xori`}
`mult`	arg1, arg2	met le produit dans `Hi` et `Lo`
`mflo`	reg	récupère la valeur de `Lo`
`div`	arg1, arg2	met le quotient dans `Lo` et le reste dans `Hi`
`j`	adresse	saute à l'adresse
`jal`	adresse	idem, enregistre l'adresse de retour dans `$ra`
`jr`	reg	saute à l'adresse contenue dans le registre
`beq`	arg1, arg2, adresse	saute si `arg1` = `arg2` ; même format pour {`bne, bgt, bge, blt, ble`}, respectivement $\{\neq, >, \geq, <, \leq\}$.
`syscall`		effectue un appel système selon la valeur de `$v0` :
	si `$v0` = 1,	affiche l'entier `$a0`,
	si `$v0` = 5,	lit un entier et le met dans `$v0`,
	si `$v0` = 10,	termine le programme.