UNIVERSITÉ LOUIS PASTEUR STRASBOURG

Master Première année
Mention Informatique

Corrigé de l'examen de Compilation de janvier 2006

Ce document constitue un corrigé synthétique de l'examen. Le premier exercice, qui reprend des éléments du cours, est laissé au soin du lecteur. L'exercice 4 correspond à une petite partie de l'examen de compilation de janvier 2005 pour lequel un corrigé détaillé a déjà été fourni. Le lecteur est donc invité à se référer à ce précédent corrigé.

Exercice 2

Considérer la grammaire suivante définissant des nombres binaires à virgule :

$$\begin{array}{lcl} S & \longrightarrow & A\ ,\ A \mid A \\ A & \longrightarrow & A\ B \mid B \\ B & \longrightarrow & 0 \mid 1 \end{array}$$

Définir un schéma de traduction en analyse ascendante qui permette de calculer la valeur décimale d'un nombre binaire à virgule. Ainsi le nombre binaire $101,101$ **a pour valeur décimale** $5,625$**. Montrer comment fonctionne ce schéma sur le nombre binaire** $1100,01$**.**

Remarque : lorsque l'on se restreint à un nombre entier et que l'on remplace $B \longrightarrow 0 \mid 1$ *par* $B \longrightarrow 0 \mid \cdots \mid 9$*, on obtient le schéma de traduction donné en cours pour l'évaluation d'un entier. La seule différence consiste ici à remplacer 10 par 2 dans le schéma. Cela correspond à la moitié de l'exercice. La seule nouveauté de cet exercice consiste à comprendre comment évaluer la partie décimale.*

Le calcul de la valeur binaire d'un décimal est donné par la formule suivante :

$$a_n a_{n-1} \cdots a_2 a_1 a_0, b_1 b_2 \cdots b_m = 2^n a_n + 2^{n-1} a_{n-1} + \cdots + 2^2 a_2 + 2^1 a_1 + 2^0 a_0 + 2^{-1} b_1 + 2^{-2} b_2 + \cdots 2^{-m} b_m$$

Dans ce schéma nous utilisons, comme dans l'exemple traité en cours, l'attribut `val` pour mémoriser la valeur décimale de la suite de bits analysée, et l'on a donc :

$$\begin{array}{lcll} B & \longrightarrow & 0 & \texttt{\{ B.val = 0 \}} \\ & \mid & 1 & \texttt{\{ B.val = 1 \}} \end{array}$$

Le traitement de la partie entière et la partie décimale étant différent, nous modifions la grammaire de la façon suivante :

$$\begin{array}{lcl} S & \longrightarrow & A\ ,\ A' \mid A \\ A & \longrightarrow & A\ B \mid B \\ A' & \longrightarrow & B\ A' \mid B \\ B & \longrightarrow & 0 \mid 1 \end{array}$$

Ainsi dans la règle $A \longrightarrow B$ le symbole B correspond au chiffre le plus à gauche et l'évaluation de la partie entière se fait en appliquant le schéma de Horner comme suit :

$$2^n a_n + 2^{n-1} a_{n-1} + \cdots + 2^2 a_2 + 2^1 a_1 + 2^0 a_0 = 2(2(\cdots(2a_n + a_{n-1}) + \cdots a_2) + a_1) + a_0$$

Le schéma de traduction est donc le suivant :

$$\begin{array}{lcll} A & \longrightarrow & A1\ B & \texttt{\{ A.val = 2 * A1.val + B.val \}} \\ & | & B & \texttt{\{ A.val = B.val \}} \end{array}$$

Dans la règle $A' \longrightarrow B$ le symbole B correspond au chiffre le plus à droite et l'évaluation de la partie décimale se fait en appliquant le schéma de Horner comme suit :

$$2^{-1}b_1 + 2^{-2}b_2 \ + \cdots 2^{-m}b_m = 2^{-1}(b_1 + 2^{-1}(b_2 + \cdots 2^{-1}(b_{m-1} + 2^{-1}b_m)\cdots))$$

Le schéma de traduction est donc le suivant :

$$\begin{array}{lcll} A' & \longrightarrow & B\ A'1 & \texttt{\{ A'.val = 2}^{-1}\texttt{(B.val + 2}^{-1}\texttt{ A'1.val) \}} \\ & | & B & \texttt{\{ A'.val = B.val \}} \end{array}$$

ou

$$\begin{array}{lcll} A' & \longrightarrow & B\ A'1 & \texttt{\{ A'.val = 2}^{-1}\texttt{(B.val + A'1.val) \}} \\ & | & B & \texttt{\{ A'.val = 2}^{-1}\texttt{ * B.val \}} \end{array}$$

Le détail du traitement de l'exemple, étant une applicaiton directe du cours et laissée à la charge du lecteur.

Exercice 3

Dans cet exercice, on considère des expressions arithmétiques quelconques, formées des opérateurs $+$ et $*$, sur lesquelles s'appliquent les règles standards de priorité et d'associativité. L'objectif de cet exercice est de déterminer pour toute expression arithmétique, son *arbre syntaxique abstrait réduit* dans lequel toutes les additions ou multiplications de même niveau sont représentées par un seul noeud, selon l'exmeple donné dans l'énoncé.

A partir de la grammaire non ambiguë des expressions arithmétiques, définir un schéma de traduction en analyse ascendante permettant de construire de tels arbres syntaxique abstraits réduits. Préciser la structure de données utilisée pour stocker cet arbre, ainsi que les opérations permettant de le manipuler.

Remarque : cet exercice était similaire à un exercice fait en cours et consistant à construire un arbre syntaxique abstrait associé à une expression arithmétique. La nouveauté de cet exercice est que l'arbre construit doit être optimisé pour avoir, le cas échéant, trois fils ou davantage associés à un noeud correspondant à une opération $+$ ou $*$.

La grammaire utilisée est la grammaire non ambigue des expressions arithématiques :

$$\begin{array}{lcl} E & \longrightarrow & E\ +\ T \mid T \\ T & \longrightarrow & T\ *\ F \mid F \\ F & \longrightarrow & (\ E\ ) \mid id \end{array}$$

Les attributs nécessaires sont les suivants :

- `X.arbre` qui désigne le pointeur sur la racine de l'arbre syntaxique abstrait réduit associé à `X`, où `X` désigne un symbole non terminal `E, T` ou `F`.
- `E.est-add`, de type booléen, qui indique si l'arbre réduit associé à `E` est de racine $+$.
- `T.est-mult`, de type booléen, qui indique si l'arbre réduit associé à `T` est de racine *.
- l'attribut `id.ptr` désigne, comme dans le cours, le pointeur sur l'entrée de la table des symboles sur `id`.

Les fonctions pour créer et manipuler un arbre syntaxique abstrait réduit sont les suivantes :

- `Creer-feuille (id.ptr)` qui crée un arbre réduit à une feuille correspondant à l'opérande `id`. Cette fonction restitue le pointeur sur l'arbre ainsi créé.

- `Creer-noeud (op,n1,n2)` où `n1` et `n2` sont deux pointeurs sur des arbres. Cette fonction crée un nouvel arbre dont la racine est l'opérateur `op` et dont les fils gauche et droit sont respectivement les arbres de racines `n1` et `n2`. Cette fonction restitue le pointeur sur l'arbre ainsi créé.
- `Ajout-fils (n, nf)` ajoute l'arbre de racine `nf` comme un nouveau fils droit de l'arbre de racine `n` et restitue le pointeur sur l'arbre `n` modifié.

La fonction `Ajout-fils` est utilisée lorsque le booléen `E.est-add` ou `T.est-mult` est à vrai, comme cela est présenté dans le schéma de traduction suivant :

```
F  ⟶  ( E )      { F.arbre = E.arbre }
   |   id        { F.arbre = Creer-feuille (id.ptr) }
T  ⟶  F          { T.arbre = F.arbre ; T.est-mult = false }
   |   T1 * F    { if T1.est-mult
                    then T.arbre = Ajout-fils ( T1.arbre, F.arbre )
                    else T.arbre = Creer-noeud ( '*',T1.arbre, F.arbre ) ;
                    T.est-mult = true }
E  ⟶  T          { E.arbre = T.arbre ; E.est-add = false }
   |   E1 + T    { if E1.est-add
                    then E.arbre = Ajout-fils ( E1.arbre, T.arbre )
                    else E.arbre = Creer-noeud ( '+',E1.arbre, T.arbre ) ;
                    E.est-add = true }
```

L'analyse de l'exemple est laissée à la charge du lecteur. Noter toutefois que le schéma, qui fonctionne bien est l'exemple donné dans l'énoncé ou d'autres exemples similaires, ne fonctionnerait pas sur une expression arithmétique "sur-parenthésée" comme $a + (b + c) + d$.

Exercice 5

L'objectif de cet exercice est de construire un schéma de traduction optimisé pour les expressions booléennes qui n'engendre qu'un quadruplet pour un test relationnel comme `A oprel B` où `oprel` désigne l'un des opérateurs relationnels $=, \neq, \leq, >$, etc. Afin de produire, pour l'expression booléenne `(((A=B) or (C=D)) and not (E=F)) or ((G=H) and (I=J))`, directement le code intermédiaire optimisé suivant, sans `goto` *inutiles* :

```
100  : if A = B goto 102          OU        100  : if A = B goto 102
101  : if C <> D goto 103                   101  : if C <> D goto 103
102  : if E <> F goto TRUE                  102  : if E <> F goto TRUE
103  : if G <> H goto FALSE                 103  : if G <> H goto FALSE
104  : if I <> J goto FALSE                 104  : if I = J goto TRUE
TRUE :                                      FALSE :
```

où `TRUE` et `FALSE` désignent l'étiquette de l'instruction élémentaire à exécuter lorsque l'expression booléenne est évaluée à `vrai`, respectivement à `false`.

Pour atteindre cet objectif, l'idée est de ne produire qu'un quadruplet lorsque l'on analyse l'expression `a oprel b` de la forme `if a oprel b goto ?`. Puis de changer l'opérateur par son "opposé" quand cela est nécessaire par le contexte que "entoure" l'expression, où l'opposé de $=$ est $<>$, celui de $\leq$ est $>$, etc.

Ainsi si l'on a l'expression composée `(a = b) or (c=d)` le code à produire est :

```
l     : if a = b goto TRUE
l+1   : if c = d goto TRUE
FALSE :
```

tandis que le code associé à `(a = b) and (c = d)` nécessite le changement en son opposé de l'opérateur $=$ de la sous-expression `(a = b)` :

```
l     : if a <> b goto FALSE
l+1   : if c = d goto TRUE
FALSE :
```

Pour réaliser cette génération, on utilise les opérations `Newtemp, Gencode, Complete` définies en cours ainsi qu'une nouvelle opération `Inverse (quad)` qui change l'opérateur contenu dans le quadruplet d'étiquette `quad` en son opposé. Cette fonction sera utilisée dans le deuxième exemple ci-dessus lorsque l'opérateur `and` sera analysé. Elle portera sur le dernier quadruplet généré, à savoir `if a = b goto ?`.

Les attributs sont ceux utilisés en cours : `E.true`, `E.false` et `oprel.val`, auquel on ajoute un nouvel attribut `E.last` qui désigne l'étiquette du dernier quadruplet associé à `E`.

La fonction booléenne `is-in (q,l)` sera utilisée pour tester si le quadruplet d'étiquette `q` est ou non dans la liste `l`.

Le schéma de traduction est le suivant. On utilise la grammaire ambiguë des expressions booléennes et l'on considère que les ambiguités sont levées, comme en Yacc, par les règles de priorités et d'associativité des opérateurs. La variable globale `nextquad` désigne l'étiquette du prochain quadruplet à générer.

```
E  ⟶  id1 oprel id2   { E.last = nextquad ;
                        E.true = Creliste ( nextquad ) ;
                        E.false = Crelist () ;
                        Gencode ( if id1.ptr oprel.val id2.ptr goto __ ) }
E  ⟶  E1 and          { if is-in ( E1.last, E1.true ) then inverse ( E1.last ) ;
                        Complete ( E1.true, nextquad ) }
      E2              { E.true = E2.true ;
                        E.false = Concat (E1.false, E2.false ) ;
                        E.last = E2.last }
E  ⟶  E1 or           { if is-in ( E1.last, E1.false ) then inverse ( E1.last ) ;
                        Complete ( E1.false, nextquad ) }
      E2              { E.true = Concat ( E1.true, E2.true ) ;
                        E.false = E2.false ) ;
                        E.last = E2.last }
E  ⟶  not E1          { E.true = E1.false ;
                        E.false = E1.true ;
                        E.last = E1.last }
E  ⟶  ( E1 )          { E.true = E1.true ;
                        E.false = E1.false ;
                        E.last = E1.last }
```

Le lecteur est invité à construire les arbres syntaxiques annotés, associés à différentes expressions booléennes, pour vérifier que le code produit est correct. Ainsi l'on obtient les traductions suivantes :

```
(A=B) or (C=D) or (E=F)        (A=B) and (C=D) and (E=F)        (A=B) and (C=D) or (E=F)

 0 : if A=B goto TRUE           0 : if A<>B goto FALSE           0 : if A<>B goto 2
 1 : if C=D goto TRUE           1 : if C<>D goto FALSE           1 : if C=D  goto TRUE
 2 : if E=F goto TRUE           2 : if E=F  goto TRUE            2 : if E=F  goto TRUE
 FALSE :                        FALSE :                          FALSE :
```

Le code généré pour l'exemple `(((A=B) or (C=D)) and not (E=F))` est :

```
   0 : if A = B goto 2
   1 : if C <> D goto FALSE
   2 : if E = F goto FALSE
TRUE :
```

Dans l'énoncé, deux codes légérement différents sont présentés, le premier avec une sortie `TRUE` à la fin, le second avec une sortie `FALSE`. Comme vous le constatez sur les exemples ci-dessus, les deux cas peuvent se présentez, selon que la dernière sous-expression est ou non une négation.

Ces deux codes peuvent avoir un intérêt, selon le type de instruction dans lequel l'expression booléenne sert de condition. Ainsi pour une instruction conditionnelle comme `if E then S1 else S2`, on privilégiera une sortie de fin à `TRUE` puisque dans le cas où l'expression booléenne est vraie, il faut exécuter l'instruction S1. Par contre, dans le cas d'une instruction du type `do S1 while E`, l'instruction `S1` doit être exécutée tant que `E` est vraie. Dans ce cas, la sortie `FALSE` en fin du code `E` sera privilégiée.