EXPRESSIONS BOOLEENNES

Instructions du code intermédiaire

(P)B[P/N/Z] $reg, etiquette

positif, négatif, zéro

JMP etiquette

Elles suffisent pour générer du code pour les structures conditionnelles et itératives

Code pour les boucles et IF

On suppose que l'expression booléenne possède un attribut place (lieu) et que les valeurs booléennes sont représentées par un octet qui vaut 1 (vrai) ou 0 (faux).

Première idée:

$I \rightarrow \ if \ E \ then \ I$

$I_1.etiquette = nouvelle.etiquette$

$I_1.code = E.code$

$\| \ gen(BZ, \ E.place, \ ',', \ I_1.etiquette)$

$\| \ I_2.code \ \| \ gen(I_1.etiquette \ )$

Pour éviter la possibilité d'avoir plusieurs étiquettes sur une instruction il faut ajouter un attribut *suit* pour l'instruction qui suit une I.

$I \rightarrow \ if \ E \ then \ I$

$I_2.suit = I_1.suit$

$I_1.code = E.code \parallel$
$gen(BZ,\ E.place,\ ',',\ I.suit)$
$\parallel I_2.code$

$I \rightarrow I \quad I$
$I_2.suit = nouvelle.etiquette$
$I_3.suit = I_1.suit$
$I_1.code = I_2.code \parallel gen(I_2.suit\ )$
$\parallel I_3.code$

Pareil pour if-then-else

Les Boucles

Code simple pour boucles **while** et **repeat** normales

```
L1 Expression.code
BZ Expression.place, L2
Instruction.code
JMP L1
L2
```

et

```
L1
Instruction.code
Expression.code
PBP Expression.place, L1
```

Le deuxième est plus court et plus efficace! Pourquoi? parce que le test est à la fin:

```
JMP L2
L1
Instruction.code
L2 Expression.code
PBP Expression.place, L1
```

est une version du code de **while** qui est plus efficace (si le nombre moyen d'itérations est supérieur à 1!)

A noter que l'idée promue par Pascal qu'il y a deux types de boucles, (à savoir celles avec test au début et continuation si le test est vrai et celles avec test à la fin et termination si les test est vrai) est erronée. (B)CPL avaient les 4 possibilités.

Evaluation d'expressions booléennes

Pour les opérateurs de comparaison:
$E \rightarrow E\ op\ E$
Prenons le cas ou op est $>$
$E_1.vrai := nouvelle.etiquette$
$E_1.code = E_2.code \parallel E_3.code \parallel$
$gen('CMP',\ \$t,\ E_2.place,\ ',',\ E_3.place)$
$gen('BP',\ \$t,\ ',',\ E_1.vrai) \parallel$
$gen('SET',\ E_1.place,\ ',',\ '0') \parallel$
$gen('JMP',\ E_1.suit) \parallel$
$gen(E_1.vrai,\ 'SET',\ E_1.place,\ ',',\ '1')$

plus les règles de calcul pour $E.suit$...

ET et OU sont comme les opérateurs arithmétiques

COMMENT FAIRE MIEUX

- Souvent le résultat d'une expression booléenne complexe est déterminé par son premier opérand

- Souvent le résultat est utilisé immédiatement pour décider le flot d'exécution.

Donc l'idée de deux attributs hérités, l'étiquette à laquelle il faut aller dans le cas d'un résultat vrai (ou faux);
Aucun besoin de représentation explicite de la valeur calculée.

exemples

$E \rightarrow E\ op\ E$

$E_1.code = E_2.code \parallel E_3.code \parallel$

$gen('CMP'\ ,\ \$t,\ ',',\ E_2.place,\ E_3.place) \parallel$

$gen('BP',\ \$t,\ ',',\ E_1.vrai) \parallel$

$gen('JMP',\ E_1.faux)$

$E \rightarrow E\ OU\ E$

$E_2.vrai = E_1.vrai$

$E_3.vrai = E_1.vrai$

$E_3.faux = E_1.faux$

$E_2.faux = nouvelle.etiquette$

$E_1.code = E_2.code \parallel gen(E_2.faux) \parallel E_3.code$

$E \rightarrow NON\ E$

$E_2.vrai = E_1.faux$

$E_2.faux = E_1.vrai$

$E_1.code = E_2.code$ aucun code pour la négation!

Les instructions switch ...

Choisir entre 4 traductions (au moins)

- suite de **if then ...**
- un tableau d'étiquettes (Fortran!) Noter l'instruction

```
GO $X, $Y, $Z
```

- un arbre binaire (interprété ou caché dans le code)
- hâchage

selon le nombre et densité des constantes (et si une fourchette de valeurs est possible).