Représentation intermédiaire

- Représentation intermédiaire
  - arbres de commandes et expression: définition
  - exemples de traduction
  - simplification
  - linéarisation
  - traces et ordonnancement

Code Intermédiaire à Arbre

Nous introduisons maintenant le premier langage intermédiaire vers lequel vous allons *traduire* notre langage source.
Il s'agit encore d'une représentation arborescente, mais dans laquelle les instructions disponibles sont beaucoup plus proches des instructions machines; on retrouve en effet:

- des étiquettes, et des sauts (conditionnels ou pas) à des étiquettes
- des accès mémoire
- des déplacements des données
- des opérations de comparaison
- des opérations arithmétiques
- l'instruction CALL

AST du Code Intermédiaire à Arbre

```
module type TREE =
sig
  type label
  type temp
  type stm = SEQ of stm * stm
           | LABEL of label
           | JUMP of exp * label list
           | CJUMP of relop * exp * exp * label * label
           | MOVE of exp * exp
           | EXP of exp
  and exp = BINOP of binop * exp * exp
          | MEM of exp
          | TEMP of temp
          | ESEQ of stm * exp
          | NAME of label
          | CONST of int
          | CALL of exp * exp list
  and binop = PLUS | MINUS | MUL | DIV | AND | OR
            | LSHIFT | RSHIFT | ARSHIFT | XOR
  and relop = EQ | NE | LT | GT | LE | GE | ULT | ULE | UGT | UGE
end
```

Commentaire

Voici une description des constructeurs Exp:

expressions (exp)

- CONST(i) l'entier i (on codera true comme 1 et false comme 0)
- NAME(n) la constante symbolique n (une étiquette assembleur)
- TEMP(t) le "registre machine" t
- BINOP(op,e,e) les opérations élémentaires
- MEM(e) la case mémoire d'adresse e (un MOVE(MEM(e),_) sera une écriture de la case mémoire d'adresse e, alors que MOVE(_,MEM(e)) sera la lecture de la case MEM(e))
- CALL(f,l) appel de la fonction f (argument évalué en premier), avec paramètres l (évalués de gauche à droite)
- ESEQ(s,e), la valeur de l'expression e après l'exécution de la commande s

Commentaire

Voici une description des constructeurs Stm:

commandes (stm)

- MOVE(TEMP(t),e) évalue e et mets le résultat dans t
- MOVE(MEM(e1,k),e2) évalue $e1$ pour obtenir une adresse mémoire a. Ensuite évalue $e2$ et place le résultat dans les k bytes à partir de a
- EXP(e) évalue e, et oublie le résultat
- JUMP(e,ls) évalue e et saute au résultat. e peut être NAME(n) ou un adresse entier. La liste ls est l'ensemble des valeurs possibles de e (optionnel, sert pour une analyse du programme)
- CJUMP(o,e1,e2,t,f) évalue e1, puis e2, et compare les résultats avec l'opérateur de comparaison o. Si $vrai$, saute à t, sinon à f
- SEQ(s1,s2) $s1$, puis $s2$
- LABEL(n) définit l'étiquette n égale à l'adresse courante

Traduction

La traduction T vers le code intermédiaire est longue, mais sans surprises. Voyons quelques cas, le reste étant laissé à completer.

Mais nous remarquons maintenant que la traduction sera effectuée en ayant accès pour toute variable simple aux attributs *level* et *offset*, et pour toute fonction à l'attribut *level*, calculés comme expliqué avant.

On supposera aussi que chaque variable occupe exactement un mot mémoire (de taille W bytes, selon la machine).
Aussi, pour les types complexes, ce mot mémoire contiendra *un pointeur* vers la structure allouée dans le tas et pas dans la pile.
Ces conventions nous permettent de nous passer de la distinction habituelle entre valeurs gauches et valeurs droits, qu'il faut quand même rappeler brièvement ici.

L-Values (valeurs gauches) et R-values (valeurs droits)

On distingue en litérature (et notamment dans les manuels C), entre deux types de valeurs:

L-values les valeurs qui peuvent apparaître à *gauche* d'une affectation, i.e. les valeurs qui désignent des cases mémoire dans lesquelles on peut écrire. En CTigre, c'est le cas de

variables simples comme x

champs d'enregistrements comme $a.nom$

éléments d'un tableau comme $a[3+5]$

R-values les valeurs qui peuvent apparaître à *droite* d'une affectation, c'est à dire les expressions qui ont une valeur que l'on peut écrire dans des cases mémoire.
En CTigre, c'est le cas de tous les L-values, mais aussi d'expressions qui ne sont pas L-values:

expressions arithmétiques comme $1+x-32$

fonctions qui retournent des types de base comme $succ(1)$

L-Values (valeurs gauches) et R-values (valeurs droits) (suite et fin)

Comme les L-values sont les plus souvent aussi des R-values, il est nécessaire de regarder le contexte pour savoir s'il faut produire du code qui *lit* une valeur depuis la case mémoire, ou du code qui *écrit* une valeur dans la case mémoire.

Dans le cas de CTigre, nous pouvons éviter cette analyse grâce à deux hypothèses simples:

- toutes les variables prennent la même place (cela se fait un forçant variables de type tableaux et enregistrement à contenir un pointeur vers la mémoire plutôt que le tableau ou enregistrement tout entier)
- la construction MEM du langage intermédiaire ne préjuge pas de la lecture ou écriture, qui est décidée par le contexte

Dans ce qui suit, la traduction d'une variable simple ou composée sera donc toujours la même sans se soucier de savoir si ell'est en position droite ou gauche.

Traduction: Variables

Le cas des variables:
La traduction de l'accès à une `SimpleVar` de *level* et *offset o* sera la suivante:

- **dans la fonction f qui la déclare (l=level(f))**:

```
MEM(BINOP(MINUS,TEMP(fp),CONST(o))
```

- **dans une fonction g englobée par f (l=level(f)<level(g))**, on suit le liens statique:

$$\texttt{MEM(BINOP(MINUS,(}\underbrace{\texttt{MEM(... MEM(}}_{\texttt{level(g)-l fois}}\texttt{ TEMP(fp))...),CONST(o))))}$$

Traduction: éléments d'un vecteur

La traduction de l'accès à un élément d'un vecteur, $e[e1]$ sera traité comme suit:

```
MEM(BINOP(PLUS,MEM(T(e)),BINOP(MULT,T(e1),CONST(W))))
```

où W est la taille d'une case mémoire (2 ou 4 bytes d'habitude), et $T(e), T(e1)$ sont les traductions de e et $e1$.

Traduction: boucles

La traduction de `while b do c done` sera

```
SEQ(LABEL(n),
    SEQ(CJUMP(EQ,T(b),CONST(1),cont,done),
        SEQ(LABEL(cont),
            SEQ(T(c),
                SEQ(JUMP(test),LABEL(done))))))
```

Traduction: boucle while

La traduction de `while b do c done` est plus simple de la visualiser de la façon suivante.

```
test:
       CJUMP(EQ,T(b),CONST(1),cont,done)
cont:  T(c)
       JUMP test
done:
```

Traduction: Appel d'une fonction

La traduction d'un appel de fonction `f(a1,...,an)` est immédiate

```
       CALL(NAME(lf),[sl,T(a1),...,T(an)])
```

Mais avec en plus le liens statique sl qui est ajouté en paramètre.
On vous rappelle que pour calculer sl il vous faut le *level* de f (connu, parce que vous l'avez déjà calculé) et celui de la fonction g qui appelle f (facile à connaître, parce-que vous êtes en train de traduire g en ce moment).

Traduction: déclaration de variable et fonction

variable La déclaration d'une variable `let a:=e in ...` produira une expression qui initialise cette variable (dans le bloc d'activation courant à *offset* connu) avec T(e).

```
MOVE(T(a),T(e))
```

fonction La déclaration d'une fonction produira une étiquette lf qui est associée à la séquence d'instructions pour le prologue, suivie de la traduction du corps de la fonction, et de l'épilogue.
Important: la traduction d'une fonction produit un nouveau Tree.stm

Traduction des chaînes de caractères

Une chaîne de caractères est normalement traduite en langage d'assemblage par une étiquette repérant l'adresse d'une directive spéciale suivie du texte de la chaîne, que l'assembleur traduira ensuite en une séquence de mots, terminés ou pas par un byte à zéro, dans une zone mémoire spéciale (généralement appelée section "data").

Dans le cas de l'émulateur SPIM, vous trouvez par exemple des fragments de code comme

```
.data
bonjour:
.asciiz "Bonjour tout le monde!\n"
```

Les opérations utilisant cette chaîne de caractères feront référence à l'étiquette assembleur `bonjour`.

Traduction: utilisation de EXP et ESEQ

Dans notre langage intermédiaire, une "expression" (type `exp`) est un arbre d'instructions qui retourne une valeur, alors que un "statement" (type `stm`) est un arbre d'instructions ne rendant pas de valeur.

Mais il y a des situations, pendant la traduction, qui nous obligent à utiliser l'un pour traduire l'autre et vice-versa.

Voyons quelques exemples d'utilisation de EXP et ESEQ:

EXP on utilise cette conversion pour les "programmes" constitués d'une expression. C'est le cas, par exemple, du programme CTigre

```
3+4
```

qui évalue l'expression mais ne fait rien de son résultat.

ESEQ on utilise cette conversion quand, pour calculer une valeur, on a besoin d'effectuer d'abord une série d'instructions.
C'est le cas de

- la traduction des opérateurs booléens
- l'initialisation de variables de type structuré (tableaux, enregistrements)

ESEQ et opérateurs booléens

En langage machine, les opérateurs de comparaison comme $\geq$, $\leq$, etc. sont souvent disponibles *seulement* pour permettre un branchement lors d'un saut, donc, on ne peut pas traduire:

```
a+b <= 32
```

par une `expression`

```
BINOP(LT, .. , ..)
```

Ce fait est bien mis en évidence dans notre langage intermédiaire par le fait que les opérateurs de relation comme LT, GT, EQ, NE etc. sont utilisables seulement dans l'instruction intermédiaire $CJUMP(op, e1, e2, t, f)$, qui saute a l'étiquette t ou f selon que l'opération *op* sur e_1 et e_2 a résultat vrai ou faux.

ESEQ et opérateurs booléens (suite et fin)

Cela nous conduit à une traduction qui nécessite l'utilisation de ESEQ.
Pour traduire

$$exp_1 \quad op \quad exp_2$$

avec *op* opérateur de comparaison, nous allons produire d'abord la séquence d'instructions

```
CJUMP(op,T(exp1),T(exp2),l0,l1)
LABEL l0
MOVE(TEMP t, CONST 1)
JUMP (NAME l2)
LABEL l1
MOVE(TEMP t,CONST 0)
LABEL l2
```

avec $l1, l2, l3$ des nouvelles étiquettes et t un nouveau nom de “registre temporaire”. À la fin de cette suite d'instructions, nous retrouvons dans le temporaire t la valeur correspondante à la comparaison (1 pour vrai et 0 pour faux). On peut alors, à l'aide de ESEQ, récupérer la valeur de t.

Très précisément, la traduction sera

```
ESEQ(
 SEQ(CJUMP(op,T(exp1),T(exp2),L0,L1),
  SEQ(LABEL L0,
   SEQ(MOVE(TEMP t,CONST 1),
    SEQ(JUMP(NAME L2),
     SEQ(LABEL L1,
      SEQ(MOVE(TEMP t,CONST 0),
       LABEL L2)))))),
 TEMP t)),
```

ESEQ et types structurés

Supposons maintenant d'avoir à traduire un fragment comme celui-ci

```
type  arrtype = array of int in
let   arr1:arrtype := arrtype [10] of 0
```

Au moment de la traduction de la variable `arr1`, nous devons produire d'abord une suite d'instructions qui allouera dans le tas l'espace mémoire nécessaire pour contenir le tableau de 10 entiers, puis initialiser chaque case à 0 et ensuite retourner comme traduction de la variable un “pointeur” sur le début de ce tableau, c'est à dire l'adresse de la première case mémoire du tableau.

À nouveau, nous avons besoin d'utiliser ESEQ, mais aussi de faire appel à une fonction externe (de système), qui sait allouer de la mémoire dans le tas. Le nom de cette fonction et la façon dont on l'appelle dépendent évidemment de la machine pour laquelle on compile (par exemple, sur le simulateur SPIM vous disposez d'un appel système `sbrk` accessible par l'instruction spéciale `syscall`).
Par simplicité, ici on utilisera le nom `malloc` et on supposera que l'on puisse appeler cette fonction comme une fonction CTigre quelconque.

ESEQ et types structurés

Alors, la traduction de

```
type  arrtype = array of int in
let   arr1:arrtype := arrtype [10] of 0
```

sera la suite d'instructions

```
ESEQ(
 SEQ(MOVE(TEMP t, CALL(NAME malloc, CONST 10*W)),
  SEQ(MOVE(MEM(BINOP(PLUS,TEMP t,CONST 0*W), CONST 0)),
      .
      .
      .
      SEQ(MOVE(MEM(BINOP(PLUS,TEMP t,CONST 9*W), CONST 0)))...)) ,
 TEMP t)
```

Remarque: dans ce code, W est la taille du mot en bytes (sur les machines modernes, c'est au moins 2, souvent 4 et quelque fois 8 (Alpha)). Aussi, les multiplications dans les CONST ici sont faites à la compilation, en connaissant W, ce n'est pas partie de la syntaxe du code intermédiaire!

De même pour les enregistrements.

Une pause de reflection

Avant de procéder, réfléchissons à ce que l'on est en train de faire: nous avons pris un programme en entrée (sous forme d'arbre de syntaxe abstraite typé et annoté avec les attributs *level* et *offset*), et nous le traduisons maintenant vers du code intermédiaire.

Lors de cette traduction, on est confrontés à un certain nombre de détails qui dépendent de la machine cible: le numéro du registre qui contient le FP (30 sur MIPS) ou le SP, la taille d'un mot en mémoire (4 sur MIPS), les conventions d'émission des chaînes de caractères, d'appel des fonctions de librairie etc. Il serait bien de garder ces dépendances groupées dans un module bien identifié.

Une pause de reflection (suite)

De plus, le résultat de cette traduction n'est pas un seul `Tree.stm`, mais plutôt *une liste* de *fragments* de code distincts:

- un fragment pour le corps du programme principal
- un fragment pour toute définition de fonction (ce fragment contient le prologue, le corps et l'épilogue de la fonction)
- un fragment pour toute chaîne de caractère dans le programme (celui ci on ne le traduira pas en Code Intermédiaire, mais on le gardera dans la liste pour la phase d'émission d'assembleur)

Une pause de reflection (fin)

Aussi, le code intermédiaire que nous produisons n'est pas, tel quel, prêt à être converti vers du langage machine.

- on souhaite pouvoir produire le code pour évaluer une expression dans n'importe lequel ordre (notamment pour minimiser le nombre de registres nécessaires pour l'évaluer). Mais certaines configurations d'un arbre de code intermédiaire font en sorte que l'ordre d'évaluation devient *significatif* en général:
  - les noeuds ESEQ dans une Tree.exp (ils peuvent produire des effets de bord)
  - des noeuds CALL dans une Tree.exp (idem)
- dans les compilateurs modernes, on renvoie souvent le (pointeur sur le) résultat dans un même registre; il est donc mieux de sauvegarder immédiatement le résultat d'une CALL dans un autre registre (sauf si ce résultat est ignoré, pour les procédures, quand CALL est fils de EXP).
- les instructions CJUMP du code intermédiaire peuvent sauter à deux adresses différents, alors que les sauts conditionnels en langage machine ont toujours l'instruction suivante comme étiquette de saut en cas de échec.

Transformations du Code Intermédiaire

On peut toujours convertir du code intermédiaire dans du code intermédiaire qui n'a pas les inconvénients cités ci-dessus.

Définition Un arbre *canonique* est un arbre de code intermédiaire tel que

- il n'y a pas de noeuds ESEQ
- le père des noeuds CALL est toujours soit un EXP soit un MOVE(TEMP t, _).

On va définir une première transformation, qui transforme tout arbre en arbre canonique.

Mise en forme canonique

Cette transformation est basée sur une idée très simple:

- on remplace tout appel de fonction CALL qui n'est pas déjà le fils d'un EXP ou un MOVE(TEMP t,_) par la séquence équivalente

```
ESEQ(MOVE(TEMP t, CALL(f,el)), TEMP t)
```

- on fait remonter tout ESEQ qui se trouve à l'intérieur d'une expression, jusqu'en haut, où le ESEQ peut se convertir en SEQ; pour cela, on peut être emmenés à introduire des nouveaux temporaires pour les expressions qui ne commutent pas avec les commandes que l'on fait remonter

Remontée des ESEQ (I)

Voyons un exemple: le fragment

```
BINOP(PLUS,
      ESEQ(s,e1),
      e2)
```

peut être remplacé par le fragment équivalent

```
ESEQ(s,
     BINOP(PLUS,
           e1,
           e2)
    )
```

Remontée des ESEQ (II)

Par contre, le fragment

```
BINOP(PLUS,
      e1,
      ESEQ(s,e2)
     )
```

peut être remplacé par le fragment

```
ESEQ(s,
     BINOP(PLUS,
           e1,
           e2)
    )
```

seulement si s et $e1$ *commutent* (i.e. si le fait d'executer s puis $e1$ ou $e1$ puis s ne change pas la sémantique du programme).
Si on ne sait pas prouver la commutation, on doit introduire un nouveau temporaire t et utiliser le fragment (toujours équivalent):

```
ESEQ(MOVE(TEMP t, e1),
     ESEQ(s,
          BINOP(PLUS,
                TEMP t,
                e2)
         )
     )
```

Remontée des ESEQ (III)

Pour toute possible configuration contenant un ESEQ à l'intérieur, on peut donner un fragment de code équivalent où l'ESEQ a soit disparu soit été remonté vers la racine.
Voici quelques autres exemples:

Le fragment	est remplacé par
MEM(ESEQ(s,e)	ESEQ(s,MEM(e))
JUMP(ESEQ(s,e)	ESEQ(s,JUMP(e))
CJUMP(op,ESEQ(s,e1),e2,t,f)	ESEQ(s,CJUMP(s,e1,e2,t,f))
MOVE(TEMP t,ESEQ(s,e))	ESEQ(s,MOVE(TEMP t,e))

Remontée des ESEQ (IV)

Et, si s et e commutent,

Le fragment	est remplacé par
BINOP(op,e1,ESEQ(s,e))	ESEQ(s,BINOP(op,e1,e))
CJUMP(op,e1,ESEQ(s,e),t,f)	ESEQ(s,CJUMP(s,e1,e,t,f))

Et s'ils ne commutent pas,

Le fragment	est remplacé par (t nouveau temporaire)
BINOP(op,e1,ESEQ(s,e))	ESEQ(MOVE(TEMP t, e1),ESEQ(s,BINOP(PLUS,TEMP t,e)))
CJUMP(op,e1,ESEQ(s,e),t,f)	ESEQ(MOVE(TEMP t, e1),ESEQ(s,CJUMP(s,TEMP t,e,t,f))

Exercice: complétez l'ensemble de règles.

Remontée des ESEQ (V)

Une fois les ESEQ arrivés en tête, on peut les remplacer par des SEQ, sans problème:

```
EXP(ESEQ(s,e))
```

devient

```
SEQ(s,EXP(e))
```

et en général,

```
EXP(ESEQ(s1,ESEQ(s2,...ESEQ(sn,e))...))
```

devient

```
SEQ(s1,SEQ(s2,...,SEQ(sn,EXP(e))...))
```

Enfin, les SEQ en tête, on peut les représenter simplement comme une liste.

Réécriture

Un ensemble de règle comme celui que l'on vient de voir donne lieu, une fois fermé par contexte, à ce que l'on appelle un *système de réécriture*, qui est étudié en informatique théorique, où l'on donne des méthode pour montrer que un ensemble de règle *termine* (i.e. le programme qui opère la transformation finit toujours) et est *confluent* (i.e. peu importe l'ordre d'application des règles, l'arbre canonique est toujours le même).
Vous pourriez en savoir plus, par exemple, en suivant les cours du DEA Programmation à Paris VII.

Remontée des ESEQ: alternative

On peut aussi remarquer que l'application de ces règles revient à extraire de tout arbre d'expression e les instructions $s1, \ldots, sn$ produisant des effets de bord et encapsulées dans des ESEQ(si,ej).
Cela produira une séquence de Tree.stm, suivie d'une Tree.exp sans effets de bord.
Au cours de cette extraction, il se peut que l'on rencontre, comme on l'a vu, des expressions qui ne commutent pas avec une instruction que l'on veut remonter, et alors on passe l'expression dans la partie instruction en introduisant un nouveau temporaire qui remplace l'expression.

Remontée des ESEQ: code de l'alternative

En suivant cette deuxième approche, on obtient le programme (dû à Appel, mais mis en Ocaml par mes soins), suivant (j'admets qu'il ne s'agit pas d'un exemple lumineux de clarté):

```
module T = Tree
let linearize (stm0: T.stm) : T.stm list =
        (* A partir d'un Tree.stm, produit une liste d'arbres canoniques t.q.
              1.  il n'y a pas de SEQ ou ESEQ
              2.  Le pere d'une CALL est un EXP(..) ou un MOVE(TEMP t,..)  *)
 let (%) a b = (* simplification: on oublie les EXP(e) dans les SEQ *)
               (* cette d\'efinition introduit un op\'erateur *INFIXE*  *)
   match (a,b) with ((T.EXP(T.CONST _)),x) -> x
   | (x, (T.EXP(T.CONST _))) -> x
   | (x,y) -> T.SEQ(x,y)

 and commute = function (* approximation minimaliste de commute *)
      (T.EXP(T.CONST _), _) -> true
    | (_, T.NAME _) -> true
    | (_, T.CONST _) -> true
    |  _ -> false

 and nop = T.EXP(T.CONST 0) (* stm trivial sans effet *)

 in let rec reorder = function
      (* transforme une liste d'exp en une liste de
```

```
        stm, puis une liste de exp sans effets de bord *)
      ((T.CALL _ as e)::rest) -> (* ici la transformation des CALL *)
        let t = Temp.new_temp()
        in reorder(T.ESEQ(T.MOVE(T.TEMP t, e), T.TEMP t) :: rest)
    | (a::rest) ->
        let (stms,e) = do_exp a
        and (stms',el) = reorder rest
        in if commute(stms',e) then (stms % stms',e::el)
           else let t = Temp.new_temp() (* si \c{c}a ne commute pas ... *)
                in (stms % T.MOVE(T.TEMP t, e) % stms', T.TEMP t :: el)
    | [] -> (nop,[])

  and reorder_exp (el,build) = let (stms,el') = reorder el
                               in (stms, build el')

  and reorder_stm (el,build) = let (stms,el') = reorder (el)
                               in stms % build(el')

  and do_stm = function (* canonicalize un stm *)
      (T.SEQ(a,b)) -> do_stm a % do_stm b
    | (T.JUMP(e,labs)) -> reorder_stm([e],fun [e] -> T.JUMP(e,labs))
    | (T.CJUMP(p,a,b,t,f)) -> reorder_stm([a;b], fun[a;b]-> T.CJUMP(p,a,b,t,f))
    | (T.MOVE(T.TEMP t,T.CALL(e,el))) ->
        reorder_stm(e::el,fun (e::el) -> T.MOVE(T.TEMP t,T.CALL(e,el)))
    | (T.MOVE(T.TEMP t,b)) -> reorder_stm([b],fun[b]->T.MOVE(T.TEMP t,b))
    | (T.MOVE(T.MEM e,b)) -> reorder_stm([e;b],fun[e;b]->T.MOVE(T.MEM e,b))
    | (T.MOVE(T.ESEQ(s,e),b)) -> do_stm(T.SEQ(s,T.MOVE(e,b))) (* ici le travail *)
    | (T.EXP(T.CALL(e,el))) -> reorder_stm(e::el,fun (e::el) -> T.EXP(T.CALL(e,el)))
    | (T.EXP e) -> reorder_stm([e],fun[e]->T.EXP e)
    |  s -> reorder_stm([],fun[]->s)

  and do_exp = function (* canonicalize une exp *)
      (T.BINOP(p,a,b)) -> reorder_exp([a;b], fun[a;b]->T.BINOP(p,a,b))
    | (T.MEM(a)) -> reorder_exp([a], fun[a]->T.MEM(a))
    | (T.ESEQ(s,e)) -> (* encore du travail *)
          let stms = do_stm s
          and  (stms',e) = do_exp e
           in (stms%stms',e)
    | (T.CALL(e,el)) -> reorder_exp(e::el, fun (e::el) -> T.CALL(e,el))
    |  e -> reorder_exp([],fun[]->e)

  (* linear elimine les SEQ en haut de l'arbre, en produisant une liste *)
  in
    let rec linear = function (T.SEQ(a,b),l) -> linear(a,linear(b,l))
                            | (s,l) -> s::l
 in (* le corps de linearize *)
    linear(do_stm stm0, [])
```

Blocs de base

Une fois notre arbre mis en forme canonique, nous nous retrouvons avec une liste d'instructions et expressions sans effets de bord, et il nous reste à traiter la deuxième différence entre code intermédiaire et assembleur: les *deux* étiquettes de CJUMP.

Nous souhaitons réordonner la liste d'instructions pour essayer de faire en sorte que la deuxième étiquette d'un CJUMP apparaisse juste après le CJUMP dans la liste (quand cela ne sera pas possible, on introduira des nouvelles étiquettes).

Blocs de base

Pour pouvoir réordonner notre code sans changer la signification du programme, nous devons d'abord le découper en *blocs de base*.

Définition: un bloc de base est une séquence d'instruction ayant les propriétés suivantes:

- la première instruction est un LABEL

- la dernière instruction est un JUMP ou un CJUMP
- il n'y a pas d'autres LABEL, JUMP ou CJUMP dans le bloc

Pour produire une liste de blocs de base à partir d'une liste d'instructions, il suffit de parcourir la liste, et créer un nouveau bloc dés que l'on rencontre un LABEL, et de le terminer dés que l'on rencontre un JUMP ou CJUMP.
Si on se trouve avec un bloc sans LABEL initial, on crée une nouvelle étiquette que l'on met en tête de ce bloc. Si on trouve un bloc qui ne se termine pas avec un JUMP ou CJUMP, on rajoute un JUMP vers l'étiquette du nouveau bloc.

Algorithme BasicBlocks

Voilà le code OCaml

```
type block = T.stm list

let basicBlocks stms =
 let donel = Temp.new_label() in
 let rec blocks =
      function
        ((T.LABEL _ as head) :: tail, blist) ->
           let rec next = function
                (((T.JUMP _) as s)::rest, thisblock)
                  -> endblock(rest, s::thisblock)
              | (((T.CJUMP _)as s )::rest, thisblock)
                  -> endblock(rest, s::thisblock)
              | ((T.LABEL lab :: _) as stms, thisblock)
                  -> next(T.JUMP(T.NAME lab,[lab]) :: stms, thisblock)
              | (s::rest, thisblock) -> next(rest, s::thisblock)
              | ([], thisblock)
                  -> next([T.JUMP(T.NAME donel, [donel])], thisblock)
           and endblock(stms, thisblock) =
               blocks(stms, List.rev thisblock :: blist)
           in next(tail, [head])
        | ([], blist) -> List.rev blist
        | (stms, blist) -> blocks(T.LABEL(Temp.new_label())::stms, blist)
  in (blocks(stms,[]), donel)
```

Traces

Une fois que nous avons notre liste de blocs de base, on peut les réarranger dans n'importe quel ordre, parce que les sauts et les étiquettes garantissent que le flot du contrôle sera toujours le bon.
On peut profiter de cette propriété pour chercher un ordre de rearrangement qui résout

notre problème de doubles étiquettes.

Définition Une *trace* est une séquence d'instructions (y compris sauts et saut conditionnels) qui peuvent être exécutés consécutivement lors de l'exécution du programme.

Un programme possède multiples traces, dont plusieurs s'intersectent, mais nous cherchons ici à construire un ensemble de traces qui couvre exactement, et sans répétition, tout le programme. Nous cherchons aussi à maximiser le nombre de JUMP ou CJUMP suivi par une des leurs étiquettes, ce qui permettra un peu d'optimisation ensuite.

Trouver les traces: un algorithme simple

Pour trouver cet ensemble de traces, en commençant avec une liste L de blocs de bases, on peut utiliser ce simple algorithme:

```
tant que L n'est pas vide faire
   initialiser une nouvelle trace vide T
   enlever l'élément de tête de L, le mettre dans a
   tant que a n'est pas marque faire
        marquer a
        ajouter a à la trace T
        examiner les successeurs de a
        s'il y a un successeur c non marque
        alors a <- c
   fin faire
   clore la trace T
fin faire
```

Étape finale

Une fois obtenu l'ensemble de traces, nous pouvons le transformer à nouveau en une liste d'instructions, et ensuite examiner cette liste pour quelques opérations de finalisation:

- tout CJUMP qui est suivi par l'étiquette correspondante à la branche false est déjà correcte
- pour tout CJUMP qui est suivi par l'étiquette correspondante à la branche true, on échange ses étiquettes true et false et on remplace le test par son complémentaire (EQ par NE, LT par GE etc.)

- pour tout CJUMP(op,e1,e2,t,f) qui n'est suivi par aucune des ses deux étiquettes t et f, on invente une nouvelle étiquette f' et on le remplace par la séquence:

```
CJUMP(op,e1,e2,t,f')
LABEL f'
JUMP(NAME f)
```

- tout JUMP suivi immédiatement par son étiquette est éliminé

Une fois cela fait, tout CJUMP est suivi par son étiquette false et on n'a pas de JUMP immédiatement suivis par leur étiquette.

Remarque: un compilateur industriel chercherait à trouver un ensemble de trace qui minimise les sauts dans les sections critiques du code (boucle, etc.).

Conclusions

Nous avons, à la fin de ce cours, pu produire une séquence d'instructions en code intermédiaire qui n'a pas d'effets de bord dans les expressions et qui est assez proche de l'assembleur pour pouvoir être convertie aisément dans une représentation intermédiaire de plus bas niveau (une sorte de assembleur abstrait), comme nous verrons dans le cours prochain.