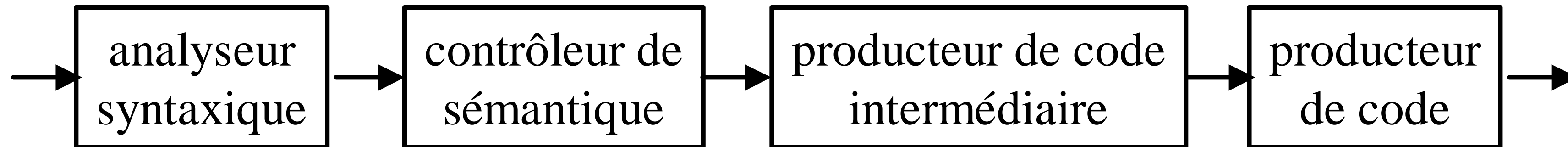
CH 5 LA PRODUCTION DE CODE

- 5.1 Les langages intermédiaires
- 5.2 Les instructions d'affectation
- 5.3 Les expressions booléennes
- 5.4 La reprise arrière

5.1 Les langages intermédiaires



Avantages : séparation de la partie frontale de la partie finale, portabilité accrue, facilité d'optimisation.

Représentation d'un programme par arbre abstrait ou par graphe acyclique orienté. (Voir traduction dirigée par la syntaxe.)

Ici, on va représenter ces arbres abstraits par un pseudo-code, le **code à trois adresses**, bien adapté aux structures de contrôle imbriquées et aux expressions algébriques. Proche d'un langage d'assemblage

Instructions `x := y op z` où `x`, `y` et `z` sont des constantes ou des noms explicites ou produits par le compilateur (temporaires) et `op` un opérateur quelconque.

Exemple :

```
a := b * - c + b * - c
```

peut être représenté par :

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
 a := t5
```

Arbre abstrait, non optimisé

```
t1 := - c
t2 := b * t1
t3 := t2 + t2
 a := t3
```

Graphe acyclique, optimisé

Temporaires = nœuds internes de l'arbre.

Instructions de plusieurs sortes, avec étiquettes symboliques si nécessaire, et pouvant contrôler le flot.

- Affectation `x := y op z`
- Affectation `x := op y`
- Copie `x := y`
- Branchement inconditionnel `goto L`
- Branchement conditionnel `if x oprel y goto L`
- Affectation avec indices `x := y [i]` et `x [i] := y`
- Pointeurs et adresses `x := & y` et `x := * y` et `* x := y`

On peut produire du code à trois adresses en utilisant un traducteur dirigé par la syntaxe.

Ici, ∥ désigne la concaténation, *NouvTemp* fabrique un nouveau temporaire et *Prod* écrit ses arguments.

Règle	Action
$I \rightarrow$ **id** **:=** E	$I.code := E.code \parallel Prod\,($**id**$.place$ '**:=**' $E.place\,)$
$E \rightarrow E + E$	$E.place := NouvTemp\,;$
	$E.code := E_1.code \parallel E_2.code \parallel$
	$Prod\,(E.place$ '**:=**' $E_1.place$ '**+**' $E_2.place\,)$
$E \rightarrow E * E$	$E.place := NouvTemp\,;$
	$E.code := E_1.code \parallel E_2.code \parallel$
	$Prod\,(E.place$ '**:=**' $E_1.place$ '*****' $E_2.place\,)$
$E \rightarrow -E$	$E.place := NouvTemp\,;$
	$E.code := E_1.code \parallel$
	$Prod\,(E.place$ '**:=**' '**UMinus**' $E_1.place\,)$
$E \rightarrow (E)$	$E.place := E_1.place\,;\ E.code := E_1.code$
$E \rightarrow$ **id**	$E.place :=$ **id**$.place\,;\ E.code :=$ ' '

Flot de contrôle pour boucle.

Règle

$I \rightarrow$ **`while`** E **`do`** I

Action

I.debut := *NouvEtiq* ; *I.apres* := *NouvEtiq* ;

I.code := *Prod* (*I. debut* ‘**`:`**’) ||

E .code ||

Prod (‘**`if`**’*E . place* ‘**`=`**’ ‘**`0`**’ ‘**`goto`**’ *I.apres*) ||

I_1 *. code* ||

Prod (‘**`goto`**’ *I.debut*) ||

Prod (*I.apres* ‘**`:`**’)

On considère dans cet exemple que le contrôle est assuré par une expression *E* qui est fausse lorsque sa valeur est 0

5.2 Les instructions d'affectation

Interaction avec la table des symboles.

Dans les instructions, au lieu d'utiliser les noms, on recherche dans la table des symboles si ce nom existe. Si oui, un pointeur vers l'entrée est retourné. Sinon, une erreur se produit.

Dans la grammaire des affectations d'expressions arithmétiques, on obtient par exemple :

$I \rightarrow$ **id :=** E { $p := Rechercher$ (**id**.nom) ;
if p **<> nil then**
$I.code := E.code \parallel Prod$ (**id**.$place$ '**:=**' $E.place$)
else $erreur$ }

$E \rightarrow E + E$ { $E.place := NouvTemp$; $E.code := E_1.code \parallel E_2.code \parallel$
$Prod$ ($E.place$ '**:=**' $E_1.place$ '**+**' $E_2.place$) }

$E \rightarrow (E)$ { $E.place := E_1.place$; $E.code := E_1.code$ }

$E \rightarrow$ **id** { $p := Rechercher$ (**id**.nom) ;
if p **<> nil then begin**
$E.place :=$ **id**.$place$; $E.code :=$' ' **end**
else $erreur$ }

Optimisation de l'utilisation des temporaires.

Encombrement des tables des symboles par la création et la non réutilisation des temporaires.

On peut les réutiliser simplement. La méthode ne convient que si l'on adopte la structure d'arbre pour les expressions arithmétiques.

On réduit $E \rightarrow E + E$.

- Si les deux E en partie droite ont comme attribut *E.place* des nom explicites, on utilise un nouveau temporaire comme *E.place* de la réduction.
- Si l'un des deux a comme attribut *E.place* un nom et l'autre un temporaire ti, celui-ci peut être réutilisé comme*E.place* de la réduction.
- Si les deux ont comme attributs des temporaires, ce sont nécessairement ti et $t < i + 1 >$. On peut donc réutiliser ti comme *E.place* de la réduction. Le futur nouveau temporaire sera donc $t < i + 1 >$.

On ne peut évidemment réutiliser des temporaires que s'ils ont le même type, ce que sait le compilateur.

Il suffit donc de conserver un compteur c qui indique quel est le nouveau temporaire à créer.
Dans le premier cas, on utilise tc ou on l'ajoute à la table des symboles et on incrémente c.
Dans le second cas, on utilise ti et on ne modifie pas c (qui vaut $i + 1$).
Dans le troisième cas, on utilise ti et on décrémente c (qui vaut à la fin $i + 1$).

On peut modifier aisément ces règles selon l'arité de l'opération.

Dans tous les cas, la valeur de c après est un de plus que le numéro du temporaire en partie gauche d'affectation.

Adressage des éléments des tableaux.

On suppose que les tableaux sont rangés par ligne et de nombre de dimensions arbitraire (cas de PASCAL).

Si le tableau contenant des éléments de taille t est déclaré `T [m1..n1, m2..n2]` et que l'adresse allouée au tableau est *base*, l'adresse de `T [i1, i2]` est

$$base + ((i1 - m1) * n2 + i2 - m2) * t \; .$$

Pour simplifier, on suppose que $m1 = m2 = 0$. Sinon, on fait un décalage de *base* par une constante.

On peut fabriquer des références à des éléments d'un tableau avec la grammaire :

$$G \rightarrow \mathbf{id}\ \mathbf{[}\ Eliste\ \mathbf{]} \mid \mathbf{id}$$

$$Eliste \rightarrow Eliste\ \mathbf{,}\ E \mid E$$

Mais les dimensions du tableau doivent être passées comme attributshérités de *Eliste*. D'où grammaire modifiée:

$$G \rightarrow Eliste\ \texttt{]} \mid \texttt{id}$$
$$Eliste \rightarrow Eliste\ \texttt{,}\ E \mid \texttt{id [}\ E$$

Ceci permet de garder la référence au tableau comme attribut synthétisé de *Eliste*. Cette référence est un pointeur vers l'entrée du tableau dans la table des symboles.

La variable E est une expression arithmétique avec la grammaire attribuée déjà vue plus haut.

Un nom susceptible d'être attribué G est repéré par deux attributs, $G.place$ et $G.depl$, le premier étant un nom dans la table des symboles et le second un décalage à partir de son adresse. Si G est in identificateur simple, $G.depl = 0$. Sinon, c'est le décalage calculé compté à partir de l'adresse de G.

On ne concatène plus le code produit, mais on l'écrit directement sur un fichier.

Affectation :

$I \rightarrow G$ **:=** E {**if** $G.depl = 0$ **then**
Prod (*G.place* '**:=**' *E.place*)
else *Prod* (*G.place* '[' *G.depl* ']' '**:=**' *E.place*) }

Arithmétique :

$E \rightarrow E + E$ { $E.place := NouvTemp$;
Prod (*E.place* '**:=**' $E_1.place$ '**+**' $E_2.place$) }

$E \rightarrow (E)$ { $E.place := E_1.place$; }

Réduction d'un nom en une expression (voir affectation) :

$E \rightarrow G$ {**if** $G.depl = 0$ **then**
$E.place := G.place$
else begin
$E.place := NouvTemp$;
Prod (*E.place* '**:=**' *G.place* '[' *G.depl* ']')
end }

Le nom est une variable simple :

$G \rightarrow$ **id** $\{$ *G.place* := **id**.*place* ; *G.depl* := 0 $\}$

Le nom est une variable indicée. Sa référence dans la table des symboles est l'attribut *.tab*. Les caractéristiques du tableau *G* sont *c* (*G.tab*), pointeur vers l'adresse de son premier élément, et la taille des éléments *taille* (*G.tab*). La taille de la dimension *m* de tableau est *limite* (*G.tab*, *m*).

$G \rightarrow$ *Eliste* **]** $\{$ *G.place* := *NouvTemp* ; *G.depl* := *NouvTemp* ;
Prod (*G.place* '**:=**' *c* (*Eliste.tab*)) ;
Prod (*G.depl* '**:=**' *Eliste.place* '*****'
taille (*Eliste.tab*)) $\}$

Eliste $\rightarrow$ *Eliste* **,** *E* $\{$ *t* := *NouvTemp* ; *m* := $Eliste_1$.*ndim* + 1 ;
Prod (*t* '**:=**' $Eliste_1$.*place* '*****' *limite* ($Eliste_1$.*tab*, *m*)) ;
Prod (*t* '**:=**' *t* '**+**' *E.place*) ; *Eliste.tab* := $Eliste_1$.*tab* ;
Eliste.place := *t* ; *Eliste.ndim* := *m* $\}$

Eliste $\rightarrow$ **id [** *E* $\{$ *Eliste.place* := *E.place* ; *Eliste.ndim* := 1 ;
Eliste.tab := **id**.*place* $\}$

Soit en récapitulant :

$I \rightarrow G$ **:=** E { **if** $G.depl = 0$ **then** $Prod\,(G.place$ '**:=**' $E.place\,)$
else $Prod\,(G.place$ '[' $G.depl$ ']' '**:=**' $E.place\,)$ }

$E \rightarrow E + E$ { $E.place := NouvTemp$;
$Prod\,(E.place$ '**:=**' $E_1.place$ '**+**' $E_2.place\,)$ }

$E \rightarrow (E)$ { $E.place := E_1.place$; }

$E \rightarrow G$ { **if** $G.depl = 0$ **then** $E.place := G.place$
else begin $E.place := NouvTemp$;
$Prod\,(E.place$ '**:=**' $G.place$ '[' $G.depl$ ']')
end }

$G \rightarrow$ **id** { $G.place :=$ **id**$.place$; $G.depl := 0$ }

$G \rightarrow Eliste$ **]** { $G.place := NouvTemp$; $G.depl := NouvTemp$;
$Prod\,(G.place$ '**:=**' $c\,(Eliste.tab)\,)$;
$Prod\,(G.depl$ '**:=**' $Eliste.place$ '*****' $taille\,(Eliste.tab)\,)$ }

$Eliste \rightarrow Eliste$ **,** E { $t := NouvTemp$; $m := Eliste_1.ndim + 1$;
$Prod\,(t$ '**:=**' $Eliste_1.place$ '*****' $limite\,(Eliste_1.tab, m)\,)$;
$Prod\,(t$ '**:=**' t '**+**' $E.place\,)$; $Eliste.tab := Eliste_1.tab$;
$Eliste.place := t$; $Eliste.ndim := m$ }

$Eliste \rightarrow$ **id [** E { $Eliste.place := E.place$; $Eliste.ndim := 1$;
$Eliste.tab :=$ **id**$.place$ }

Supposons que **`T`** soit un tableau de dimension 10 x 20 et **`U`** un tableau 5 x 15 et que tous les éléments soient entiers ($t = 4$), l'instruction **`T [a + b, c ] := U [a, b + c]`** produit le code à trois adresses suivant, où **`baseT`** et **`baseU`** sont les adresses de **`T`** et de **`U`** :

```
       t1 := a + b
       t2 := t1 * 20
       t2 := t1 + c
       t3 := baseT
       t4 := t2 * 4
       t5 := b + c
       t6 := a * 15
       t6 := t6 + t5
       t7 := baseU
       t8 := t6 * 4
       t9 := t7 [t8]
  t3 [t4] := t9
```

5.3 Les expressions booléennes

Importance particulière, car servent à gouverner le flot de contrôle.

Grammaire utilisée pour els expressions booléennes :

$E \to E$ **or** E | E **and** E | **not** E | (E) | **id oprel id** | **true** | **false**

Représentation numérique (1 = **true**, 0 = **false**)

La traduction en code à trois adresses ne pose pas de problème par rapport à la traduction des expressions arithmétiques, au moyen d'une grammaire S-attribuée.

Instructions du flot de contrôle.

$I \rightarrow$ **if** E **then** I | **if** E **then** I **else** I | **while** E **do** I

A une expression booléenne qui contrôle un branchement, on associe deux étiquettes symboliques *E.vrai* et *E.faux*. Une étiquette supplémentaire permet de sortir du **if then else**. Elles sont implémentées comme attributs hérités.

$I \rightarrow$ **if** E **then** I { *E.vrai* := *NouvEtiq* ; *E.faux* := *I.suivant* ;
I_1*.suivant* := *I.suivant* ;
I.code := *E.code* || *Prod* (*E.vrai* '**:** ') || I_1*.code* }

$I \rightarrow$ **if** E **then** I **else** I { *E.vrai* := *NouvEtiq* ; *E.faux* := *I.suivant* ;
I_1*.suivant* := *I.suivant* ; I_2*.suivant* := *I.suivant* ;
I.code := *E.code* || *Prod* (*E.vrai* '**:** ') || I_1*.code* ||
Prod ('**goto**' *I.suivant*) || *Prod* (*E.faux* '**:** ') || I_2*.code* }

$I \rightarrow$ **while** E **do** I { *I.debut* := *NouvEtiq* ; *E.vrai* := *NouvEtiq* ;
E.faux := *I.suivant* ; I_1*.suivant* := *I.debut* ;
I.code := *Prod* (*I.debut* '**:** ') || *E.code* ||
Prod (*E.vrai* '**:** ') || I_1*.code* || *Prod* ('**goto**' *I.debut*) }

Traduction du flot de contrôle des expressions booléennes : Utilisation d'attributs hérités pour produire une évaluation paresseuse.

$E \rightarrow E$ **or** E	$\{E_1.vrai := E.vrai \ ; E_1.faux := NouvEtiq \ ;$ $E_2.vrai := E.vrai \ ; E_2.faux := E.faux \ ;$ $E.code := E_1.code \parallel Prod\ (E_1.faux$ '**:**' $) \parallel E_2.code \}$
$E \rightarrow E$ **and** E	$\{E_1.vrai := NouvEtiq \ ; E_1.faux := E.faux \ ;$ $E_2.vrai := E.vrai \ ; E_2.faux := E.faux \ ;$ $E.code := E_1.code \parallel Prod\ (E_1.vrai$ '**:**' $) \parallel E_2.code \}$
$E \rightarrow$ **not** E	$\{E_1.vrai := E.faux \ ; E_1.faux := E.vrai \ ;$ $E.code := E_1.code \}$
$E \rightarrow (E)$	$\{E_1.vrai := E.vrai \ ; E_1.faux := E.faux \ ;$ $E.code := E_1.code \}$
$E \rightarrow$ **id oprel id**	$\{ E.code := Prod\ ($'`if`' **id** $_1.place$ **oprel**$.op$ **id**$_2.place$ '`goto`' $E.vrai) \parallel$ $Prod\ ($'`goto`' $E.faux) \}$
$E \rightarrow$ **true**	$\{ E.code := Prod\ ($'`goto`' $E.vrai) \}$
$E \rightarrow$ **false**	$\{ E.code := Prod\ ($'`goto`' $E.faux) \}$

Exemple : `while a < b or c < d and e < f do`
`if g < h then x := y + z else x := y - z`

```
L1 : if a < b goto L2
     goto L3
L3 : if c < d goto L4
     goto Lf
L4 : if e < f goto L2
     goto Lf
L2 : if g < h goto L5
     goto L6
L5 : t1 := y + z
     x := t1
     goto L1
L6 : t2 := y - z
     x := t2
     goto L1
Lf :
```

Les attributs hérités des expressions booléennes empêchent une évaluation ascendante, même avec marqueurs.

Solution possible : évaluation en deux passes, après construction de l'arbre syntaxique, et calcul récursif de l'attribut *.code* (ou écriture sur le fichier de sortie).

Branchements à choix multiples.

Le test de branchement ne peut être produit qu'à la fin, lorsque sont connues les diverses valeurs du branchement.

```
case E of
  begin
     V1  : I1
...
     Vn  : In
  other : Id
  end
```

```
        < code qui évalue E dans t >
        goto test
   L1 : < code pour I1 >
        goto suite
...
   Ln : < code pour In >
        goto suite
   Ld : < code pour Id >
        goto suite
 test : if t = V1 goto L1
...
        if t = Vn goto Ln
        goto Ld
suite :
```

5.4 La reprise arrière

Problème rencontré lors de l'évaluation paresseuse des expressions booléennes : on n'a pas une grammaire S-attribuée.

Raison : on ne connaît pas encore l'étiquette de l'instruction à laquelle on doit se rendre : celle-ci dépend d'éléments qui seront analysés plus tard.

Ce problème peut se rencontrer dans d'autres circonstances.

Solution possible : effectuer plusieurs passes avec évaluation récursive des attributs.

On va présenter une autre technique, dite de **reprise arrière**. On laisse les branchements provisoirement indéfinis et on les reprend lorsque l'adresse de branchement est connue.

Hypothèse sur le code à trois adresses utilisé :
chaque instruction a un numéro qui lui sert d'étiquette.

Les instructions possédant des directions en suspens sont gérées comme des listes d'étiquettes :

- *CreerListe* (*i*) crée une nouvelle liste contenant *i* et retourne un pointeur vers cette liste ;
- *Fusion* (p_1, p_2) concatène les deux listes pointées par p_1 et p_2 et retourne un pointeur vers le résultat ;
- *ReprendreArriere* (*p*, *i*) insère *i* comme étiquette de direction dans toutes les instructions de la liste pointées par *p*, puis désalloue l'espace occupé par la liste pointée par *p*.
- Une variable globale *numero* contient le numéro de la prochaine instruction produite.

La grammaire utilisée est S-attribuée

$E \to E$ **or** $M\ E$	{ *ReprendreArriere* (E_1.*faux* , *M.num*) ; *E.vrai* := *Fusion* (E_1.*vrai*, E_2.*vrai*) ; *E.faux* := E_2.*faux* }
$E \to E$ **and** $M\ E$	{ *ReprendreArriere* (E_1.*vrai* , *M.num*) ; *E.vrai* := E_2.*vrai* ; *E.faux* := *Fusion* (E_1.*faux*, E_2.*faux*) }
$E \to$ **not** E	{ *E.faux* := E_1.*vrai* ; *E.vrai* := E_1.*faux* }
$E \to$ (E)	{ *E.vrai* := E_1.*vrai* ; *E.faux* := E_1.*faux* }
$E \to$ **id oprel id**	{ *E.vrai* := *CreerListe* (*numero*) ; *E.faux* := *CreerListe* (*numero* + 1) ; *Prod* ('`if`' **id** $_1$.*place* **oprel**.*op* **id**$_2$.*place* '`goto`') ; *Prod* ('`goto`') }
$E \to$ **true**	{ *E.vrai* := *CreerListe* (*numero*) ; *Prod* ('`goto`') }
$E \to$ **false**	{ *E.vrai* := *CreerListe* (*numero*) ; *Prod* ('`goto`') }
$M \to \varepsilon$	{ *M.num* := *numero* }

Dans l'exemple **`a < b or c < d and e < f`** on obtient à la fin de l'analyse :

```
100 : if a < b goto
101 : goto 102
102 : if c < d goto 104
103 : goto
104 : if e < f goto
105 : goto
```

E.vrai vaut {100, 104}
E.faux vaut {103, 105}.
Les destinations de ces instructions seront complétées ultérieurement.

On utilise maintenant cette technique pour la traduction du flot contrôle en une passe.

$I \rightarrow$ **if** E **then** I | **if** E **then** I **else** I | **while** E **do** I | **begin** L **end** | A
$L \rightarrow L$ **;** I | I

où L est une liste d'instructions et A une instruction d'affectation.

Pour la traduction, on utilise deux listes *L.suiv* et *I.suiv* qui contiennent les numéros de toutes les instructions contenant un branchement vers l'instruction qui suit *L* ou *I*.

Cas du **while do** :

$I \rightarrow$ **while** $M\ E$ **do** $M\ I$ { *ReprendreArriere* ($E.vrai$, $M_2.num$) ;
ReprendreArriere ($I_1.suiv$, $M_1.num$) ;
Prod ('`goto`' $M_1.num$) ;
$I.suiv := E.faux$ }

Cas de **if then else** :

$I \rightarrow$ **if** E **then** $M\ I\ N$ **else** $M\ I$ { *ReprendreArriere* ($E.vrai$, $M_1.num$) ;
ReprendreArriere ($E.faux$, $M_2.num$) ;
$I.suiv :=$ *Fusion* ($I_1.suiv$, $N.suiv$, $I_2.suiv$) }

Le marqueur *N* sert à sauter par-dessus le deuxième *I*.

$N \rightarrow \varepsilon$ { $N.suiv :=$ *CreerListe* (*numero*) ;
Prod ('`goto`') }

L'affectation initialise *I.suiv* et produit le code convenable (voir plus haut code pour affectations) :

$I \rightarrow A$ { *I.suiv* := *CreerListe* ( ) ;
     Prod (*x* '**=**' *y*) }

Pour la réduction en une liste :

$L \rightarrow L$ **;** $M\ I$ {*ReprendreArriere* (L_1.*suiv*, *M.num*);
     L.suiv := *I.suiv* }

On obtient en récapitulant la grammaire suivante :

$I \rightarrow$ **if** E **then** $M\ I$	{ *ReprendreArriere* (*E.vrai*, *M.num*) ; *I.suiv* := *Fusion* (*E.faux*, I_1*.suiv*) }
$I \rightarrow$ **if** E **then** $M\ I\ N$ **else** $M\ I$	{ *ReprendreArriere* (*E.vrai*, M_1*.num*) ; *ReprendreArriere* (*E.faux*, M_2*.num*) ; *I.suiv* := *Fusion* (I_1*.suiv*, *N.suiv*, I_2*.suiv*) }
$I \rightarrow$ **while** $M\ E$ **do** $M\ I$	{ *ReprendreArriere* (*E.vrai*, M_2*.num*) ; *ReprendreArriere* (I_1*.suiv*, M_1*.num*) ; *Prod* ('`goto`' M_1*.num*) ; *I.suiv* := *E.faux* }
$M \rightarrow \varepsilon$	{ *M.num* := *numero* }
$N \rightarrow \varepsilon$	{ *N.suiv* := *CreerListe* (*numero*) ; *Prod* ('`goto`') }
$I \rightarrow$ **begin** L **end**	{ *I.suiv* := *L.suiv* }
$I \rightarrow A$	{ *I.suiv* := *CreerListe* ( ) ; *Prod* (*x* '`=`' *y*) }
$L \rightarrow L$ **;** $M\ I$	{*ReprendreArriere* (L_1*.suiv*, *M.num*); *L.suiv* := *I.suiv* }

De la sorte, le fragment :

```
if a < b or c < d and e < f then
     x := 1
     else begin
          x := 0 ;
          u := 1
     end ;
while a < b do x := x + 1 ;
```

produit le code attendu :

```
100 : if a < b goto 106
101 : goto 102
102 : if c < d goto 104
103 : goto 108
104 : if e < f goto 106
105 : goto 108
106 : x := 1
107 : goto 110
108 : x := 0
109 : u := 1
110 : if a < b goto 112
111 : goto
112 : t1 := x + 1
113 : x := t1
114 : goto 110
```

Le résultat est la variable L dont $L.suiv$ vaut $\{111\}$.

Les étiquettes explicites (**goto etiq**) sont traitées comme des identificateurs et placées dans la table des symboles, dont les numéros sont laissés en blanc. Tant qu'on a un **goto etiq** on laisse la destination en blanc en gérant une liste des instructions dirigeant sur **etiq**. Lorsque l'instruction étiquetée **etiq** est rencontrée, on complète la table des symboles et on fait une reprise arrière sur les étiquettes de la liste associée à **etiq**. Par la suite, les **goto etiq** peuvent être traités directement par la table des symboles.

Ceci permet d'appliquer aux étiquettes les règles de portée de tout identificateur.