Chapitre 8
Génération de code intermédiaire

1. Langages intermédiaires
2. Déclarations
3. Instructions d'affectation
4. Expressions booléennes
5. Appels de procedures

Introduction

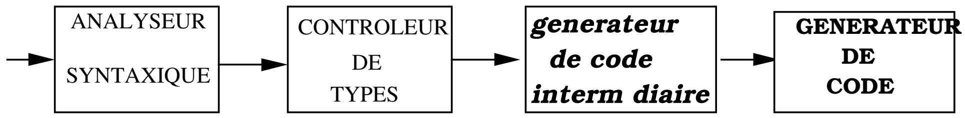


Figure 1: place du générateur de code intermédiaire

Constitue la fin de la partie frontale (‘*front-end*’) du compilateur: elle est indépendante de la machine cible.

Avantages:

1. Possibilité de redirection vers des machines différentes.
2. Possibilité d’optimisation du code plus grandes.

Peut être incorporé éventuellement à l’analyseur syntaxique (ascendant ou descendant).

Langages intermédiaires

Forme abstraite: arbres syntaxiques. Grammaire

Figure 2: arbre syntaxique de a:=b*-c + b*-c

attribuée pour le calcul de l'arbre syntaxique d'une instruction d'affectation.

REGLE	ACTION SEMANTIQUE
$S \rightarrow \mathbf{id}\ :=\ E$	$S.nptr := mknode(':=',$ $mkleaf(\mathbf{id}, \mathbf{id}.place),$ $E.nptr)$
$E \rightarrow E + E$	$E.nptr := mknode('+',$ $E_1.nptr, E_2.nptr)$
$E \rightarrow E * E$	$E.nptr := mknode($ $'*', E_1.nptr, E_2.nptr)$
$E \rightarrow -E$	$E.nptr := mknode($ $'umoins', E.nptr)$
$E \rightarrow (E)$	$E.nptr := E_1.nptr$
$E \rightarrow \mathbf{id}$	$E.nptr := mkleaf(\mathbf{id},$ $\mathbf{id}.place)$

Code à trois adresses

Proche d'un assembleur. On utile des noms symboliques et des instructions de contrôle. Forme générale des instructions:

`x` := `y` *op* `z`

où *op* est un opérateur. Chaque instruction réfère à trois adre! sses.

Liste des instructions.

1. Instructions d'affectation:`x` := `y` *op* `z` ou `x` := *op* `y`
2. Instructions de saut inconditionnel: `goto L` vers l'instruction d'étiquette `L` ou de saut conditionnel `if x` *relop* y goto L
3. Instructions indicées `x:=y[i]` et `x[i]:=y`

Traduction dirigée par la syntaxe: utilise des noms temporaires pour l'évaluation des expressions. Une expression aura comme attributs

1. $E.place$ qui est l'identificateur contenant la valeur de E.
2. $E.code$ qui est la suite des instructions du code à trois adresses.

$S \rightarrow \mathbf{id} := E$	$S.code := E.code\|$
	$gen(\mathbf{id}.place' :=' E.place)$
$E \rightarrow E + E$	$E.place := newtemp;$
	$E.code := E_1.code\|$
	$E_2.code\|gen(E.place :=$
	$E_1.place' +' E_2.place$
$E \rightarrow E * E$	$E.place := newtemp;$
	$E.code! := E_1.code\|$
	$E_2.code\|gen(E.place :=$
	$E_1.place' *' E_2.place$
$E \rightarrow -E$	$E.place := newtemp;$
	$E.code := E_1.code\|$
	$gen(E.place' :='' uminus'$
	$E_1.place)$
$E \rightarrow (E)$	$E.place := E_1.place;$
	$E.code := E_1.code;$
$E \rightarrow \mathbf{id}$	$E.place := \mathbf{id}.place$
	$E.code :=''$

Pour les instructions de contrôle, on aura des schémas du type suivant:

REGLE	ACTION
$S \rightarrow \mathbf{while} E \mathbf{do} S$	$S.begin := newlabel;$
	$S.after := newlabel;$
	$S.code := gen(S.begin$
	$' :') \| E.code \|$
	$gen('if' E.place' ='' 0'$
	$'goto' S.after) \|$
	$S_1.code \|$
	$gen('goto' S.begin) \|$
	$gen(S.after' :')$

L'implémentation des suites d'instructions en code à trois adresses peut se faire par quadruplets

$op, arg1, arg2, resultat$

ou par triplets

$op, arg1, arg2$

en utilisant les noms des triplets comme nom du résultat.

Exempl! e:

	op	$arg1$	$arg2$	$result$
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

	op	$arg1$	$arg2$
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	affect	a	(4)

On peut bien entendu utiliser des listes chaînées des triplets constituant des triplets dits *indirects*, avec des avantages évidents pour la manipulation (optimisation).

Déclarations

Déclarations groupées en début de procédure comme en Pascal.

Utilisation d'une adresse relative: on utilise une variable globale $offset$.

Chaque type T se voit associer deux attributs: *type* (une expression de type construite à partir des types de base) et *width* ! (pour prendre en compte la place mémoire occupée par les objets de ce type).

Schéma de traduction:

$P \rightarrow$		$\{offset := 0\}$
	D	
$D \rightarrow$	$D\ ;\ D$	
$D \rightarrow$	$\mathbf{id} : T$	$\{enter(\mathbf{id}.name,$ $offset); offset :=$ $offset + T.width\}$
$T \rightarrow$	**integer**	$\{T.type := integer;$ $T.width := 4\}$
$T \rightarrow$	**real**	$\{T.type := real;$ $T.width := 8\}$
$T \rightarrow$	**array** $[\mathbf{num}]\mathbf{of}\ T$	$\{T.type := array($ $\mathbf{num}.val, T_1.type);$ $T.width := \mathbf{num}.val \times$ $T_1.width\}$
$T \rightarrow$	$\uparrow T$	$\{T.type := pointer($ $T_1.type); T.width := 4\}$

La première règle:

$$P \rightarrow \{offset := 0\} D$$

peut être transformée en ajoutant un marqueur en:

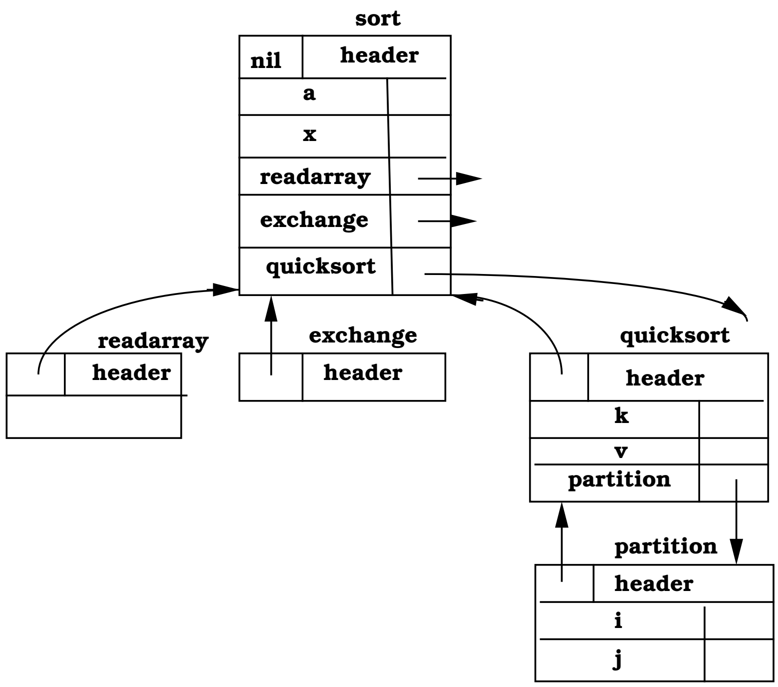
$$\begin{array}{l} P \rightarrow M\ D \\ M \rightarrow \epsilon \qquad \{offset := 0\} \end{array}$$

de façon à écrire l'action en fin de règle.

Pour prendre en c! ompte la portée des déclarations, on considère une grammaire plus complète:

$$\begin{array}{rl} P \rightarrow & D \\ D \rightarrow & D \mid \mathbf{id} : T \mid \mathbf{proc\ id}\ ; D ; S \end{array}$$

On suppose qu'il y a une table de symboles séparée pour chaque procédure.



On utilise les opérations suivantes:

1. $mktable(previous)$ crée une nouvelle table de symboles et rend un pointeur dessus. Un pointeur vers la procédure englobante $previous$ est ajouté.

2. $enter(table, name, type, offset)$ crée une entrée de plus.

3. $addwidth(table, width)$ garde la somme cumulée des tailles des entrées de la table.

4. $enterproc(table, name, newtable)$ crée une nouvelle entrée pour une procédure $name$ avec $newtable$ pointant sur la nouvelle table de symboles.

On utilise aussi deux piles:

tblptr pour garder un pointeur sur la table de la procédur! e en cours de traitement

offset pour tenir à jour l'offset cumulé de la procédure en cours de traitement.

Schéma de traduction:

$$
\begin{array}{lll}
P \rightarrow & MD & \{addwidth(top(tblptr), \\
 & & top(offset)); \\
 & & pop(tblptr); pop(offset)\} \\
M \rightarrow & \epsilon & \{t := mktable(nil); \\
 & & push(t, tblptr); \\
 & & push(0, offset)\} \\
D \rightarrow & D; D & \\
D \rightarrow & \textbf{proc id}; & \\
 & N\ D; S & \{t := top(tblptr); \\
 & & addwidth(t, top(offset)) \\
 & & pop(tblptr); pop(offset); \\
 & & enterproc(top(tblptr), \\
 & & \textbf{id}.name, t)\} \\
D \rightarrow & \textbf{id} : T & \{enter(top(tblptr), \\
 & & \textbf{id}.name, T.type, \\
 & & top(offset)); top(offset := \\
 & & top(offset) + T.width\} \\
N \rightarrow & \epsilon & \{t := mktable(top(tblptr)); \\
 & & push(t, tblptr); \\
 & & push(0, offset)\}
\end{array}
$$

En ajoutant la règle:

$$T \rightarrow \textbf{record}\ L\ D\ \textbf{end}$$

on peut utiliser les types enregistrements. les actions sont très semblables à celles des appels de p! rocédure:

$$
\begin{array}{lll}
T \rightarrow & \textbf{record} & \\
 & L\ D & \\
 & \textbf{end} & \{T.type := \\
 & & record(top(tblptr)); \\
 & & T.width := top(offset); \\
 & & pop(tblptr); pop(offset)\}
\end{array}
$$

$$
\begin{array}{lll}
L \rightarrow & \epsilon & \{t := mktable(nil); \\
 & & push(t, tblptr); \\
 & & push(0, offset)\}
\end{array}
$$

Affectations

Le schéma suivant permet de créer et utiliser les noms créés pour traduire les affectations.

Il utilise les attributs **id**.name pour le lexeme représenté par un identificateur et une fonction $lookup(\mathbf{id}.name)$ pour consulter la table des symboles.

$$
\begin{array}{ll}
S \rightarrow \mathbf{id} := E & \{p := lookup(\mathbf{id}.name); \\
& \mathbf{if}\, p \neq nil \,\mathbf{then} \\
& emit(p\ ' :=' \ E.place) \\
& \mathbf{else}\, error\} \\
\\
E \rightarrow E + E & \{E.place := newtemp; \\
& emit(E.place\, ' :=' \ E_1.place \\
& ' +' \ E_1.place)\} \\
\\
E \rightarrow E * E & \{E.place := newtemp; \\
& emit(E.place\ ' :=' \ E_1.place \\
& ' *' \ E!_1.place)\} \\
E \rightarrow -E & \{E.place := newtemp; emit \\
& (E.place\ ' :=' \ \ ' -' \ E_1.place)\} \\
\\
E \rightarrow (E) & \{E.place := E_1.place\} \\
E \rightarrow \mathbf{id} & \{p := lookup(\mathbf{id}.name); \\
& \mathbf{if}\, p \neq nil \,\mathbf{then} \\
& E.place := p \\
& \mathbf{else}\, error\}
\end{array}
$$

Si on ajoute la possibilité de procédures en faisant apparaitre une instruction d'affectation selon la grammaire:

$$\begin{array}{ll} P \rightarrow & M\ D \\ M \rightarrow & \epsilon \\ D \rightarrow & D; D \ \mid \ \mathbf{id} : T \ \mid \ \mathbf{proc\ id}; N\ D; S \\ N \rightarrow & \epsilon \end{array}$$

Il faut raffiner la fonction $lookup$ de recherche dans la table des symboles en une recherche d'abord dans la table de la procédure courante (en utilisant la pile $tblptr$) puis dans la procédure englobante, etc.

Adressage des éléments de tableau

L'adresse de $a[i]$ est

$$base + (i - low) \times w$$

où low est la borne ! inférieure des indices, $base$ est l'adresse relative du début du tableau et w est la taille des éléments du tableau. Cette expression peut aussi s'écrire:

$$i \times w + (base - low \times w)$$

où le deuxième terme peut être évalué à la compilation.

En langage C, $a[i]$ est équivalent à $*(a + i)$ et les tableaux commencent à zero. L'expression ci-dessus se réduit donc à

$$i \times w + base$$

où $base$ correspond à l'adresse de a.

Tableaux à plusieurs dimensions

Ils peuvent être rangés:

1. Ligne par ligne ('row major') comme en Pascal et C
2. Colonne par colone ('column-major') comme en Fortran

La formule pour l'adresse de $a[i_1, i_2]$ ligne par ligne est

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

où n_2 est le nombre des valeurs possibles de i_2.

Cette expression peut s'écrire:

$$\begin{aligned} &(\quad (\quad i_1 \times n_2) + i_2) \times w \\ &\qquad + (base - ((low_1 \times n_2) + low_2) \times w) \end{aligned}$$

où, comme précédemment, le deuxième terme peut être calculé à la compilation.

Plus généralement, la représentation ligne-par-ligne donne comme adresse de $a[i_1, i_2, \ldots, i_k]$ l'expression:

$$\begin{aligned} &(\quad (\quad \cdots ((i_1 n_2 + i_2) n_3 + i_3) \cdots) n_k + i_k) \times w \\ &\qquad + base - (\cdots ((low_1 n_2 + low_2) n_3 + low_3) \cdots) n_k + low_k) \times w \end{aligned}$$

Le deuxième terme peut, de nouveau être calculé à la compilation et rangé dans la table des symboles à l'entrée de a.

En langage C, on écrit $a[i_1][i_2]$ à la place de $a[i_1, i_2]$. Comme de plus tous les tableaux commencent à 0, le deuxième terme de la formule ci-dessus se réduit à $base$.

Pour traduire les références de tableau, on utilise la récurrence:

$$\begin{aligned} e_1 &= i_1 \\ e_m &= e_{m-1} \times n_m + i_m \end{aligned}$$

Si on utilise la grammaire:

$$\begin{aligned} L &\rightarrow \mathbf{id}[\, Elist \,] \mid \mathbf{id} \\ Elist &\rightarrow Elist, E \mid E \end{aligned}$$

l'attribut donnant la borne n_m est hérité (il est attaché au nom du tableau). Pour le transformer en attribut synthétisé, on transforme la grammaire en:

$$\begin{aligned} L &\rightarrow Elist \,] \mid \mathbf{id} \\ Elist &\rightarrow Elist, E \mid \mathbf{id}[\, E \end{aligned}$$

On peut maintenant passer un pointeur vers l'entrée de la table de symboles pour le tableau comme attribut synthétisé $Elist.array$ de $Elist$.

Le schéma de traduction complet part de la grammaire:

$$S \rightarrow L := E$$

$$\begin{array}{rcl} E & \to & E+E \\ E & \to & (E) \\ E & \to & L \\ L & \to & Elist\,] \\ L & \to & \mathbf{id} \\ Elist & \to & Elist, E \\ Elist & \to & \mathbf{id}[\,E \end{array}$$

On utilise deux attributs pour une l-valeur L: $L.place$ et $L.offset$. Si L est un nom simple, $L.offset$ vaut **null** et $L.place$ est un pointeur vers la table des symboles.

On a d'abord:

$$\begin{array}{ll} S \to L := E & \{\mathbf{if}\, L.offset = \mathbf{null\ then} \\ & emit(L.place' :=' E.place; \\ & \mathbf{else} \\ & emit(L.place'['L.offset']'' :=' E.place)\} \end{array}$$

Quand un élément de tableau est réduit en une expression, on prend sa r-valeur:

$$\begin{array}{ll} E \to L & \{\mathbf{if}\, L.offset = \mathbf{null\ then} \\ & E.place := L.place \\ & \mathbf{else\ begin} \\ & E.place := newtemp; \\ & emit(E.place' :=' L.place'['L.offset']')\mathbf{end}\} \end{array}$$

Le calcul de $L.offset$ donne ensuite:

$$\begin{array}{ll} L \rightarrow Elist\,] & \{L.place := newtemp; \\ & L.offset := newtemp; \\ & emit(L.place' :=' constant(Elist.array)); \\ & emit(L.offset' :=' w' *' Elist.place)\} \end{array}$$

Et dans le cas d'un simple identificateur:

$$\begin{array}{ll} L \rightarrow \mathbf{id} & \{L.place := \mathbf{id}.place; \\ & L.offset := \mathbf{null}\} \end{array}$$

Le calcul par récurrence utilise un attribut $ndim$ pour le nombre de dimensions de $Elist$:

$$\begin{array}{ll} Elist \rightarrow Elist, E & \{t := newtemp; \\ & m := Elist_1.ndim + 1; \\ & emit(t' :=' Elist_1.place' *' \\ & \quad limit(Elist_1.array, m)); \\ & emit(t' :=' t' +' E.place); \\ & Elist.array := Elist_1.array; \\ & Elist.place := t; \\ & Elist.ndim := m\} \end{array}$$

Enfin:

$$\begin{array}{ll} Elist \rightarrow \mathbf{id}\,[\,E & \{Elist.place := E.place; \\ & Elist.ndim := 1; \\ & Elist.array := \mathbf{id}.place\} \end{array}$$

Exemple

Soit a un tableau $[1..10] \times [1..20]$ avec $w = 4$. Traduction de `x:=a[y,z]`:

$$
\begin{aligned}
t_1 &:= y * 20 \\
t_1 &:= t_1 + z \\
t_2 &:= a - 84 \;\; (\text{Base de a} - 84) \\
t_3 &:= 4 * t_1 \\
t_4 &:= t_2[t_3] \\
x &:= t_4
\end{aligned}
$$

Expressions booléennes et redirections

Les expressions booléennes servent au calcul de valeurs logiques en particulier pour les instructions de contrôle.

Grammaire pour la syntaxe:

$$\begin{array}{rcl} E & \rightarrow & E\,\textbf{or}\;E \mid E\,\textbf{and}\;E \mid \textbf{not}E \mid (E) \\ & & \mid \textbf{id relop id} \mid \textbf{true} \mid \textbf{false} \end{array}$$

Méthodes de traduction:

1. Comme les expressions arithmétiques.
2. Par redirection

Représentation numérique

On évalue complétement l'expression comme une expression arithmétique.

Par exemple l'expression `b or x < y` sera traduite:

```
100: t := b
101: if x < y goto 104
102: u := 0
103: goto 105
104: u := 1
105: v := t or u
```

Redirections

On traduit une expression booléenne par une redirection sans nécessairement l'évaluer en entier (*short-circuit code*). Pour cela on associe à une expression booléenne E deux étiquettes $E.vrai$ et $E.faux$: ce sont les étiquettes indiquant où le contrôle doit être transféré suivant que $E=$ vrai ou faux. $E.vrai$ et $E.faux$ sont des attributs

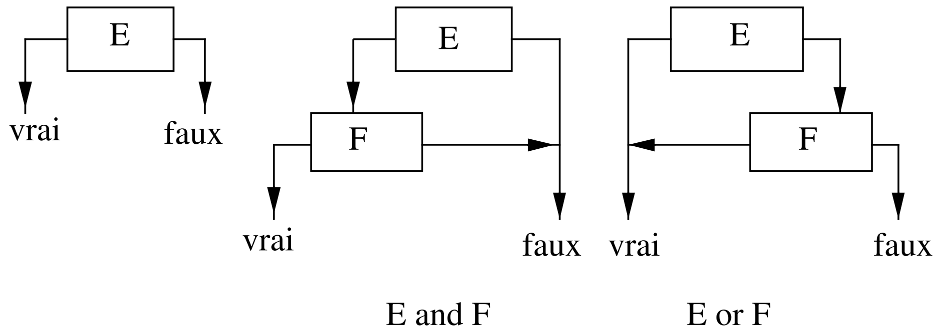


Figure 3: Représentation graphique

hérités.

Exemple

L'expression

`a < b or c < d and e < f`

sera traduite par

```
    if a < b goto Evrai
    goto E1
E1: if c < d goto E2
    goto Efaux
E2: if e < f goto Evrai
    goto Efaux
```

Instructions de contrôle

Grammaire:

$$
\begin{array}{rcl}
S & \rightarrow & \textbf{if } E \textbf{ then } S \\
  & | & \textbf{if } E \textbf{ then } S \textbf{ else } S \\
  & | & \textbf{while } E \textbf{ do } S
\end{array}
$$

On utilise un attribut hérité $S.suivant$ qui désigne l'étiquette de l'instruction suivant la traduction de S et un attribut $S.debut$ pour le début.

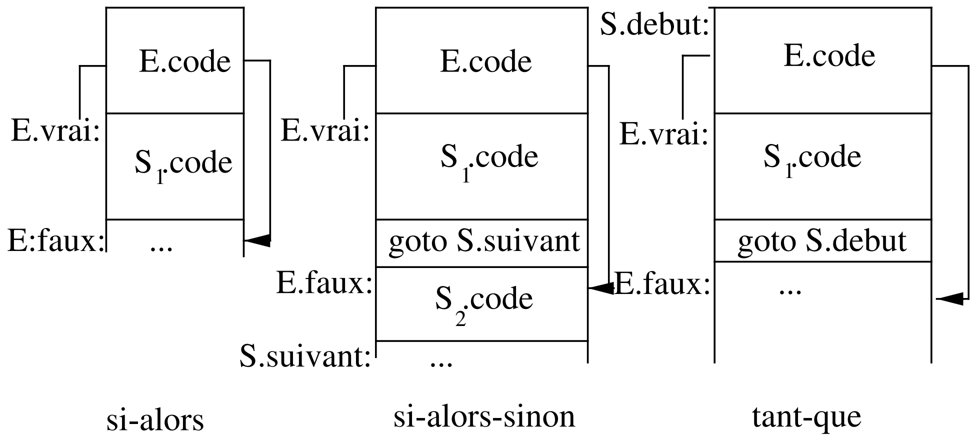


Figure 4: Traduction des redirections

Implémentation

Difficulté: on ne connait pas les adresses des instructions au moment où les goto sont engendrés (i.e. la grammaire n'est pas L-attribuée). Deux possibilités:

1. Opérer en deux passes (ou laisser des étiquettes symboliques dans le code intermédiaire)
2. Utiliser la reprise arrière (ou *backpatching*).

Nous décrivons une implémentation de la reprise arrière. On utilise une traduction en quadruplets et trois fonctions:

1. $makelist(i)$ crée une nouvelle liste de quadruplets contenant i et retourne un pointeur sur cette liste.
2. $merge(p, q)$ concatène les deux listes p et q et retourne un pointeur sur la liste créée.
3. $backpatch(p, i)$ donne à tous les quadruplets de la liste p le but i.

Les attributs $E.vrai$ et $E.faux$ deviennent des listes de quadruplets contenant les goto vers les étiquettes correspondantes. Ils deviennent *synthétisés*

On utilise la grammaire avec marqueurs:

$$\begin{aligned} E \rightarrow \quad & E\,\mathbf{or} M\ E \\ & \mid E\ \mathbf{and}\ M\ E \\ & \mid \mathbf{not} E \\ & \mid (E) \\ & \mid \mathbf{id\ relop\ id} \\ & \mid \mathbf{true} \\ & \mid \mathbf{false} \\ M \rightarrow \epsilon & \end{aligned}$$

Le marqueur M sert à enregistrer la valeur $M.q$ du quadruplet courant. Cette valeur est conservée dans une variable *nextquad* mise à jour par la fonction *emit* qui produit un quadruplet.

Schéma de traduction pour les expressions booléennes

$E \rightarrow E$ **or** $M\ E$	$\{backpatch(E_1.f, M.q);$ $E.v := merge(E_1.v, E_2.v);$ $E.f := E_2.f\}$
$E \rightarrow E$**and**$M\ E$	$\{backpatch(E_1.v, M.q);$ $E.v := E_2.v;$ $E.f := merge(E_1.f, E_2.f)\}$
$E \rightarrow$ **not**E	$\{E.v := E_1.f; E.f := E_1.v\}$
$E \rightarrow (E)$	$\{E.v := E_1.v; E.f := E_1.f\}$
$E \rightarrow$ **id relop id**	$\{E.v := mklist(nextquad);$ $E.f := mklist(nextquad + 1);$ $emit('if'$**id**$.place$ **relop**$.op$ **id**$.place$`'goto ?'`$);$ $emit($`'goto ?'`$)\}$
$E \rightarrow$ **true**	$\{E.v := makelist(nextquad);$ $emit($`'goto ?'`$)\}$
$E \rightarrow$ **false**	$\{E.f := makelist(nextquad);$ $emit($`'goto ?'`$)\}$
$M \rightarrow \epsilon$	$\{M.q := nextquad\}$

La traduction est

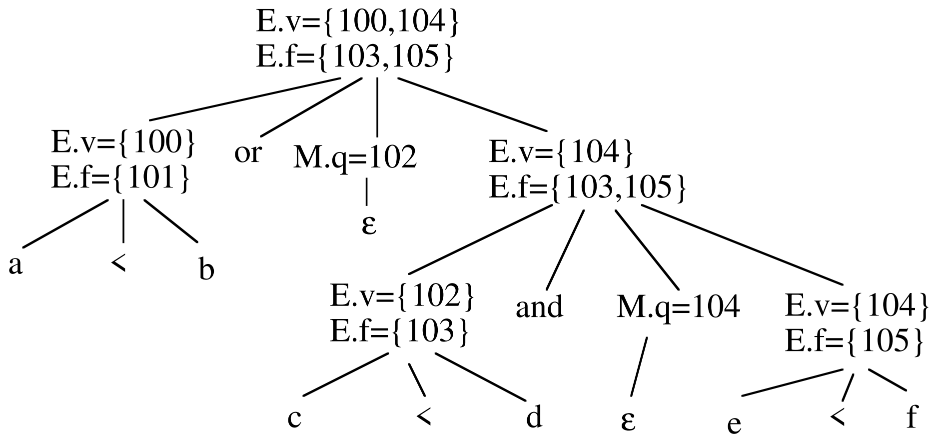


Figure 5: Arbre décoré de a < b or c < d and e < f

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

Instructions de controle

On utilise la grammaire:

$$\begin{aligned} S \rightarrow\ & \textbf{if } E \textbf{ then } S \mid \textbf{if } E \textbf{ then } S \textbf{ else } S \\ & \mid \textbf{while } E \textbf{ do } S \mid \textbf{begin } L \textbf{ end} \\ & \mid A \\ L \rightarrow\ & L; S \mid S \end{aligned}$$

où L désigne une suite d'instructions et A une instruction d'affectation.

On crée les attributs $S.suivant$ et $L.suivant$ pour contenir la liste des quadruplets visant (par un goto) le quadruplet qui suit S ou L. On modifie la grammaire en introduisant un marqueur M pour conserver la valeur du quadruplet courant et N pour sauter à la suite dans une instruction conditionnelle complète:

$$\begin{aligned} S \rightarrow\ & \textbf{while } M\ E \textbf{ do } M\ S \\ S \rightarrow\ & \textbf{if } E \textbf{ then } M\ S\ N \textbf{ else } M\ S \\ M \rightarrow\ & \epsilon \\ N \rightarrow\ & \epsilon \end{aligned}$$

Schéma de traduction

On aura d'abord en utilisant le marqueur M:

$$M \rightarrow \epsilon \ \{M.q := nextquad\}$$

les traductions:

$$
\begin{aligned}
S \rightarrow \textbf{if} \quad & E \ \textbf{then} \ M \ S \\
& \{backpatch(E.v, M.q); \\
& S.suivant := merge(E.f, S_1.suivant)\}
\end{aligned}
$$

$$
\begin{aligned}
S \rightarrow \textbf{while} \quad & M \ E \ \textbf{do} \ M \ S \\
& \{backpatch(S_1.suivant, M_1.q); \\
& backpatch(E.v, M_2.q) : \\
& S.suivant := E.f; \\
& emit(\texttt{'goto'} M_1.q)\}
\end{aligned}
$$

Pour l'instruction conditionnelle complète on utilise le marqueur N:

$$\begin{aligned} N \to \epsilon\ \ & \{N.suivant := makelist(nextquad); \\ & emit(\texttt{'goto ?'})\} \end{aligned}$$

$$\begin{aligned} S \to \textbf{if}\ \ & E\ \textbf{then}\ M\ S\ N\ \textbf{else}\ M\ S \\ & \{backpatch(E.v, M_1.q); \\ & backpatch(E.f, M_2.q); \\ & S.suivant := merge(S_1.suivant, \\ & N.suivant, S_2.suivant)\} \end{aligned}$$

Le reste ne pose pas de difficulté:

$$\begin{array}{ll} S \to \textbf{begin}\ L\ \textbf{end} & \{S.suivant := L.suivant\} \\ S \to A & \{S.suivant := makelist()\} \\ L \to L; M\ S & \{backpatch(L.suivant, M.q); \\ & L.suivant := S.suivant\} \\ L \to S & \{L.suivant := S.suivant\} \end{array}$$

Exemple

Traduction de

```
while (a < b) do
 if (c < d) then x := y+z
```

On obtient:

100: if a<b goto 102
101: goto 107
102: if c<d goto 104
103: goto 100
104: t := y+z
105: x:=t
106: goto 100
107: