Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Compilation
Licence d'informatique 3e année
Éric Laporte

Expressions booléennes
Structures de contrôle

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Expressions booléennes

E --> *E* **or** *E* | *E* **and** *E* | **not** *E* | **(** *E* **)** | **id relop id** | **true** | **false**

```
if (index<MAX &&
   list[index]!=value )
```

Les expressions booléennes sont souvent employées dans la condition d'une structure de contrôle : `if`, `while`, `for`

Évaluation stricte (*eager evaluation*)

On évalue toutes les sous-expressions

Évaluation paresseuse (*short-circuit evaluation*)

On n'évalue une sous-expression que si c'est indispensable

`&&` et `||` en C, Java, C++

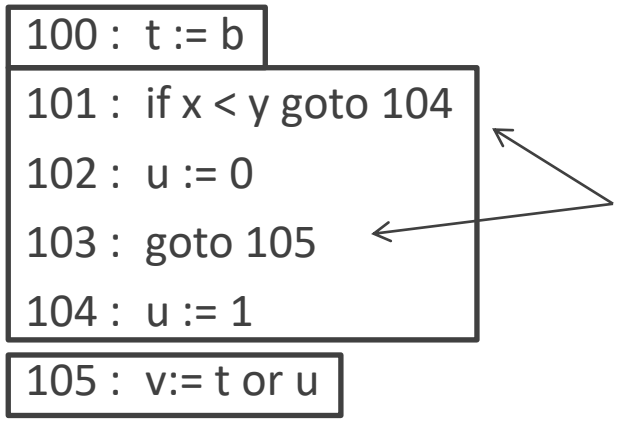
`and then` et `or else` en Ada

Évaluation stricte (*eager evaluation*)

Code source

b or x < y

Code intermédiaire



Comme les expressions arithmétiques
0 représente faux, 1 représente vrai
Sauvegarder les valeurs des sous-expressions dans la pile ou dans des variables temporaires
Si b vaut vrai, on exécute quand même les instructions 101 et 105
Sauts en avant : le compilateur écrit les instructions cibles après avoir écrit les sauts
- si on utilise des étiquettes symboliques, on n'a pas besoin de savoir où sera l'instruction cible
- si on utilise des étiquettes numériques, on peut calculer la cible

`&` et `|` en C, Java, C++
`and` et **`or`** en Ada

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Évaluation stricte

E --> *E* **or** *E*

| *E* **and** *E*

| **not** *E*

| **(** *E* **)**

| **id relop id**

| **true**

| **false**

Exercice : écrire en pseudocode un traducteur par parcours de l'arbre abstrait

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

```
E --> E or E
    | E and E
    | not E
    | ( E )
    | id relop id
    | true
    | false
```

Traduction des expressions booléennes
Évaluation stricte

Nœud **Or**

Or.place := newtemp(); visit(**Or**.firstchild); visit(**Or**.secondchild);
write(**Or**.place ':=' **Or**.firstchild.place 'or' **Or**.secondchild.place) ;

Nœud **And**

And.place := newtemp(); visit(**And**.firstchild); visit(**And**.secondchild);
write(**And**.place ':=' **And**.firstchild.place 'and' **And**.secondchild.place);

Nœud **Not**

Not.place := newtemp(); visit(**Not**.firstchild);
write(**Not**.place ':= not' **Not**.child.place) ;

Nœud **relop**

iftrue := newlabel(); iffalse := newlabel(); **relop**.place := newtemp() ;
visit(**relop**.firstchild); visit(**relop**.secondchild);
write('if' **relop**.firstchild.place **relop**.op **relop**.secondchild.place 'goto' iftrue);
write(**relop**.place ':= 0'); write('goto' iffalse) ;
write(iftrue ':' **relop**.place ':= 1'); write (iffalse ':');

```
101 : if x < y goto 104
102 : u := 0
103 : goto 105
104 : u := 1
```

Évaluation paresseuse (*short-circuit evaluation*)

```
if (index<MAX &&
   list[index]!=value )
```

Avantages

Éviter une erreur dans la sous-expression court-circuitée

Rapidité si l'évaluation qu'on court-circuite est couteuse : appel de fonction

Ordre d'évaluation

Le résultat dépend de l'ordre dans lequel le programme évalue les sous-expressions

C, Java, C++ garantissent l'évaluation de gauche à droite pour les opérateurs booléens

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Évaluation paresseuse : premier essai

Code source

b or x < y

Code intermédiaire

```
100 :  t := b
101 :  if t=1 goto 105
102 :  if x < y goto 105
103 :  u := 0
104 :  goto 106
105 :  u := 1
106 :
```

Si b vaut vrai, on n'exécute pas l'instruction 102

Comparaison

Code source

b or x < y

Code intermédiaire

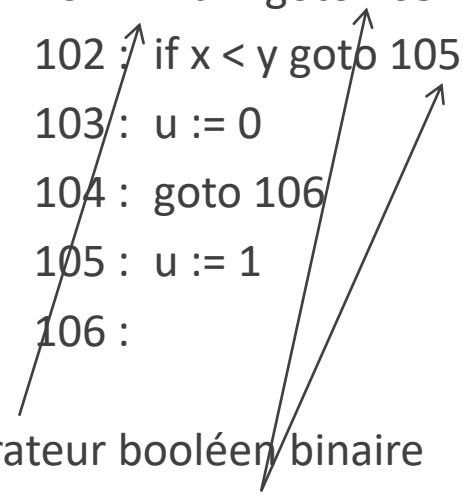
Évaluation stricte

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

Code intermédiaire

Évaluation paresseuse

100 : t := b
101 : if t=1 goto 105
102 : if x < y goto 105
103 : u := 0
104 : goto 106
105 : u := 1
106 :



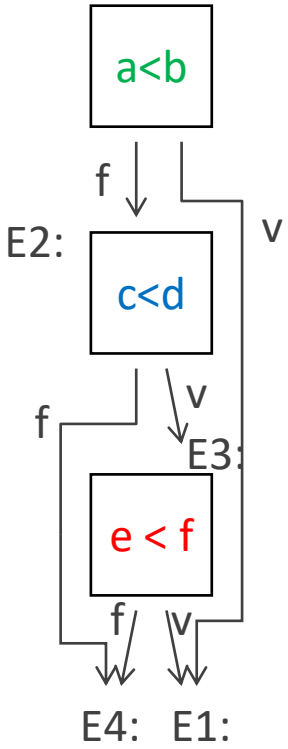
Un saut pour chaque opérateur booléen binaire

Plusieurs sauts vers la même étiquette

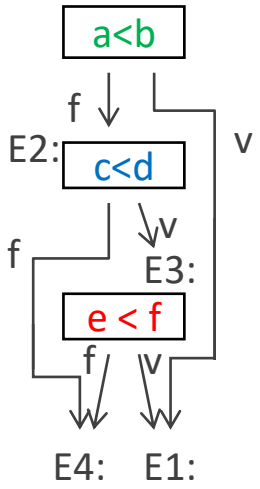
Évaluation paresseuse par sauts

Code source

a < b or c < d and e < f



On ne sauvegarde aucune valeur booléenne avant la fin de l'expression complète



Évaluation paresseuse par sauts

Code source

a < b or c < d and e < f

Code intermédiaire

```
        if a < b goto E1
        goto E2
E2:     if c < d goto E3
        goto E4
E3:     if e < f goto E1
        goto E4
E1:     (...)
        (...)
E4:     (...)
```

Plusieurs sauts vers la même étiquette

Sauts en avant

Réalisation

Avec étiquettes symboliques

Avec étiquettes numériques :

- on ne connait leur valeur que quand on écrit les instructions cibles
- donc il faut modifier des sauts déjà écrits (reprise arrière, *backpatching*)

Avec étiquettes symboliques

Code source

a < b or c < d and e < f

Code intermédiaire

```
        if a < b goto E1
        goto E2
E2:     if c < d goto E3
        goto E4
E3:     if e < f goto E1
        goto E4
E4:     (...)
        (...)
E1:     (...)
```

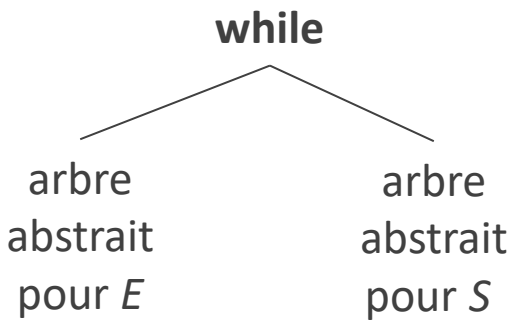
On peut utiliser une étiquette symbolique sans savoir où sera l'instruction cible

Dans un champ *E*.true, on met l'étiquette symbolique cible pour les cas où l'évaluation de *E* donne "vrai"

Avantage des étiquettes symboliques

On ne change pas une instruction déjà écrite

Avec étiquettes symboliques



Nœud **while**

```
begin := newlabel() ; after := newlabel() ;
iftrue := newlabel() ;
while.firstchild.true := iftrue ;
while.firstchild.false := after ;
write(begin ':') ;
visit(while.firstchild);
write(iftrue ':') ;
visit(while.secondchild) ; write('goto' begin);
write(after ':') ;
```

On initialise les champs *E*.true et *E*.false avant de parcourir *E*

Avec étiquettes symboliques

Nœud **Or**	tmp := newlabel() ; **Or**.firstchild.true := **Or**.true ; **Or**.firstchild.false := tmp ; **Or**.secondchild.true := **Or**.true ; **Or**.secondchild.false := **Or**.false ; visit(**Or**.firstchild); write(tmp ':') ; visit(**Or**.secondchild);
Nœud **And**	tmp := newlabel() ; **And**.firstchild.true := tmp ; **And**.firstchild.false := **And**.false ; **And**.secondchild.true := **And**.true; **And**.secondchild.false := **And**.false; visit(**And**.firstchild); write(tmp ':') ; visit(**And**.secondchild);
Nœud **Not**	**Not**.child.false := **Not**.true ; **Not**.child.true := **Not**.false ;

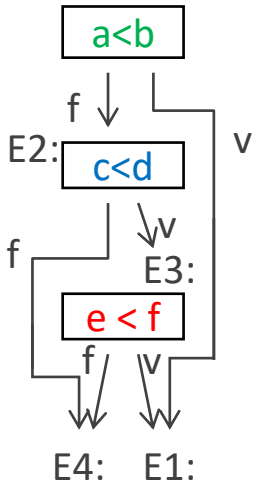
Nœud **relop**	visit(**relop**.firstchild); visit(**relop**.secondchild);
	write('if' **relop**.firstchild.place **relop**.op **relop**.secondchild.place 'goto' **relop**.true);
	write('goto' **relop**.false) ;

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Grammaire attribuée pour traduction ascendante

E	--> *E* **or** *M E*	*M*.label :=E_1.f ; *E*.v := E_1.v ; *E*.f := E_2.f ; write(E_2.v, ':'); write('goto', *E*.v) ;
M	--> ε	write(*M*.label, ':');
E	--> *E* **and** *M E*	*M*.label :=E_1.v ; *E*.v := E_2.v ; *E*.f := E_1.f ; write(E_2.f, ':'); write('goto', *E*.f) ;
E	--> **not** *E*	*E*.v := E_1.f ; *E*.f := E_1.v ;
E	--> **(** *E* **)**	*E*.v := E_1.v ; *E*.f := E_1.f ;
E	--> **id relop id**	*E*.v := newlabel() ; *E*.f := newlabel() ; write('if', **id**$_1$.place, **relop**, **id**$_2$.place, 'goto', *E*.v) ; write('goto', *E*.f) ;
E	--> **true**	*E*.v := newlabel() ; *E*.f := newlabel() ; write('goto', *E*.v) ;
E	--> **false**	*E*.v := newlabel() ; *E*.f := newlabel() ; write('goto', *E*.f) ;



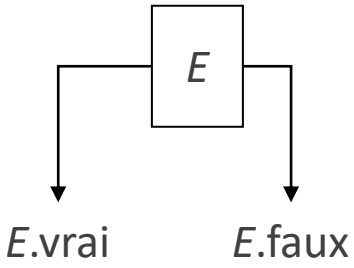
Avec étiquettes numériques et reprise arrière (*backpatching*)

Code source

a < b or c < d and e < f

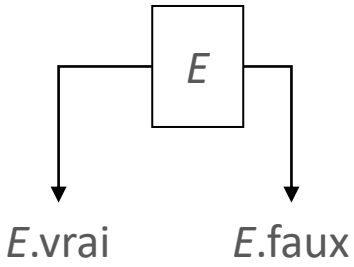
Code intermédiaire

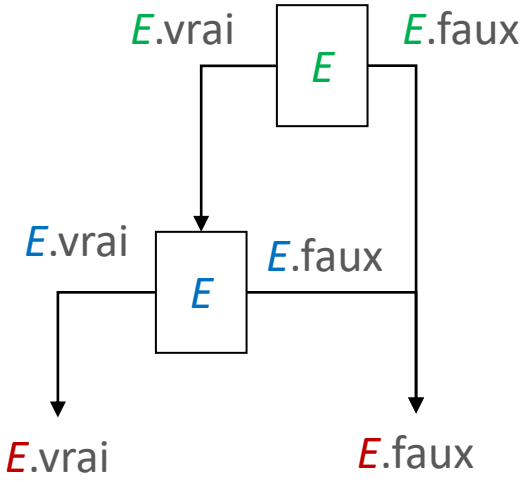
```
        if a < b goto E1
        goto E2
E2:     if c < d goto E3
        goto E4
E3:     if e < f goto E1
        goto E4
E1:     ...
E4:     ...
```



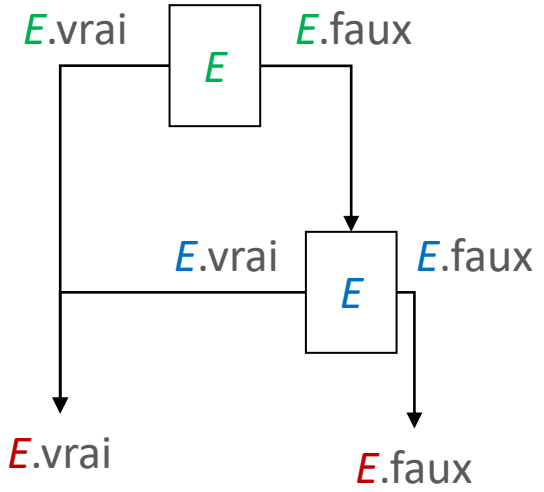
- Méthode compatible avec la traduction ascendante
- Donne le même résultat qu'avec le parcours de l'arbre abstrait
- Dans un attribut *E*.vrai, on met la liste des sauts vers l'instruction cible
- Quand on écrit ces sauts, on les laisse incomplets
- Quand on connaît le numéro de l'instruction cible, on les complète

Avec étiquettes numériques et reprise arrière





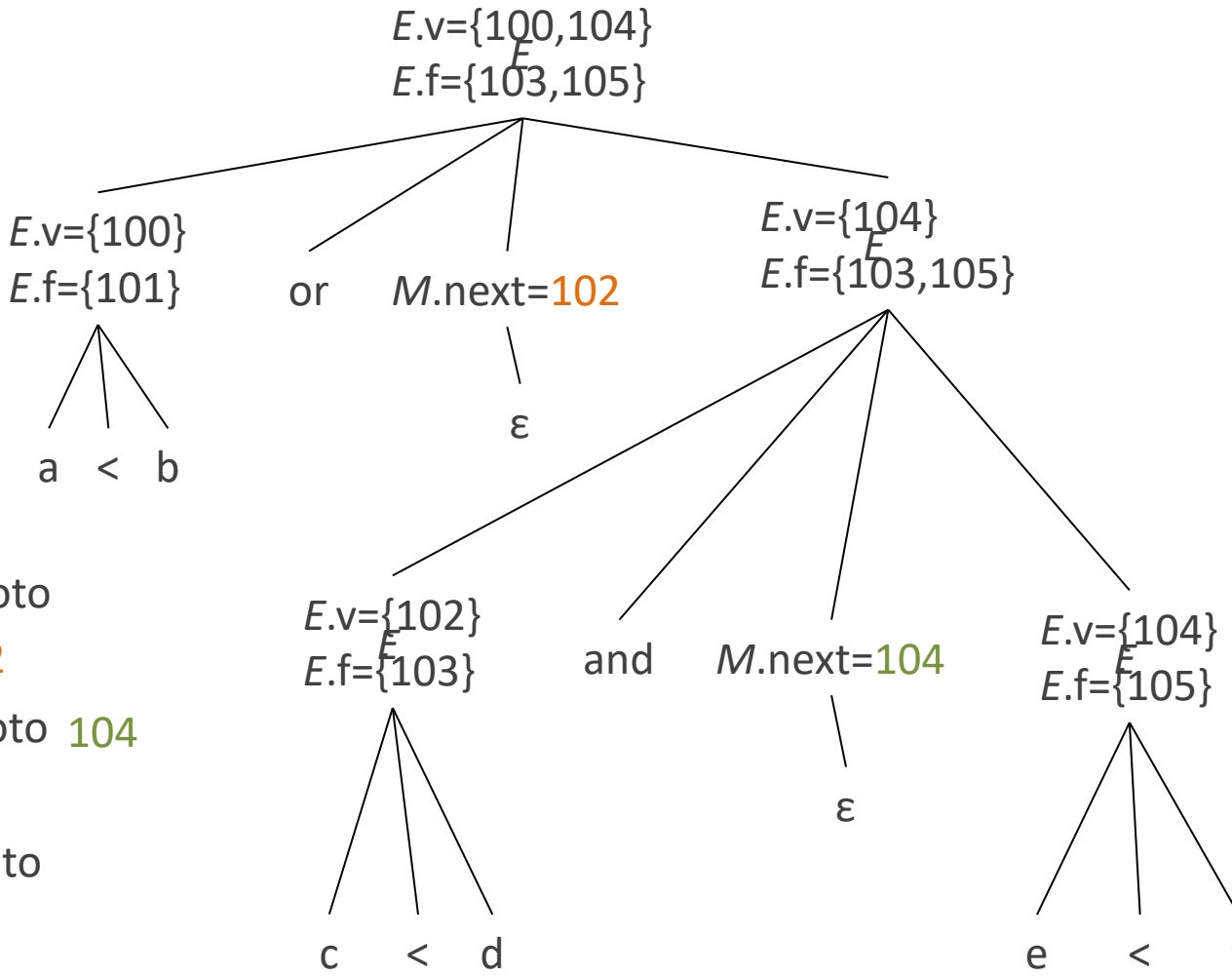
E --> *E* **and** *E*



E --> *E* **or** *E*

E.vrai : liste des sauts vers l'instruction cible pour le cas où *E* est vrai

Exemple



100 : if a < b goto

101 : goto 102

102 : if c < d goto 104

103 : goto

104 : if e < f goto

105 : goto

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Grammaire attribuée

E --> *E* **or** *E*
E --> *E* **and** *E*
E --> **not** *E*
E --> (*E* **)**
E --> **id relop id**
E --> **true**
E --> **false**

makeList(i) crée une nouvelle liste chaînée d'instructions de saut contenant seulement i

merge(p, q) concatène deux listes chaînées d'instructions

backpatch(p, i) complète les instructions de saut contenues dans la liste p en leur donnant pour cible l'instruction i

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Grammaire attribuée

E	--> *E* **or** *M E*	backpatch(E_1.f, *M*.next) ; *E*.f := E_2.f ;
		E.v := merge(E_1.v, E_2.v) ;
M	--> ε	*M*.next = next ;
E	--> *E* **and** *M E*	backpatch(E_1.v, *M*.next) ; *E*.v := E_2.v ;
		E.f := merge(E_1.f, E_2.f) ;
E	--> **not** *E*	*E*.v := E_1.f ; *E*.f := E_1.v ;
E	--> **(** *E* **)**	*E*.v := E_1.v ; *E*.f := E_1.f ;
E	--> **id relop id**	*E*.v := makeList(next) ;
		write('if', **id**$_1$.place, **relop**, **id**$_2$.place, 'goto') ;
		E.f := makeList(next) ; write('goto') ;
E	--> **true**	*E*.v := makeList(next) ; write('goto') ;
E	--> **false**	*E*.f := makeList(next) ; write('goto') ;

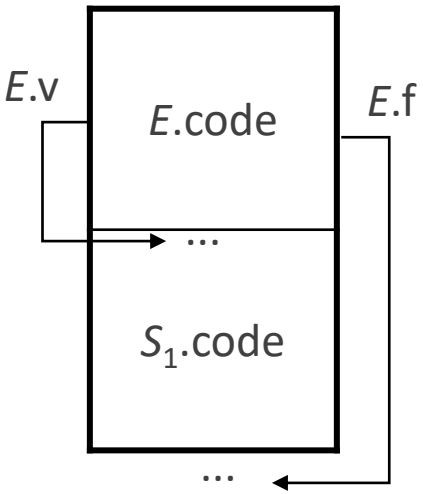
Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

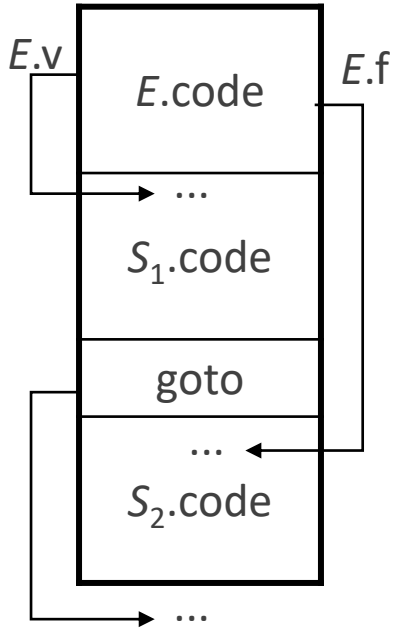
Sommaire

Structures de contrôle

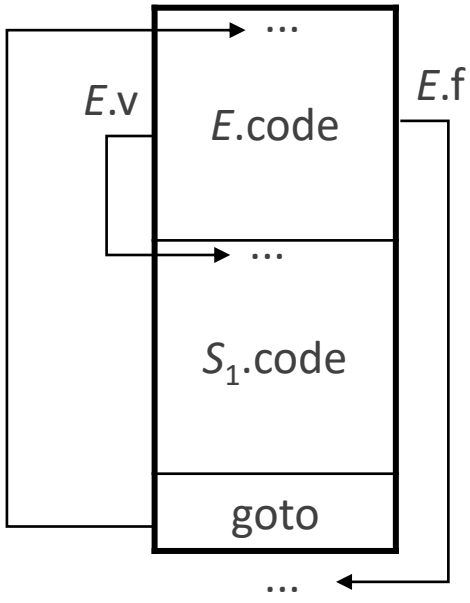
S --> **if (** *E* **)** *S* | **if (** *E* **)** *S* **else** *S* | **while (** *E* **)** *S*



si-alors



si-alors-sinon



tant que

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Évaluation paresseuse par sauts

S --> **if (** *E* **)** *S* | **if (** *E* **)** *S* **else** *S* | **while (** *E* **)** *S* | **{** *L* **}** | *A*

L --> *L* **;** *S* | *S*

A représente une affectation

Une instruction *S* peut contenir des sauts vers la première instruction à exécuter après *S*

Exemple : un saut dans la condition d'un **while**

De même pour *L*

Exemple : si le dernier élément de la liste est un **while**

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Évaluation paresseuse par sauts avec étiquettes symboliques

S --> **if (** *E* **)** *S* | **if (** *E* **)** *S* **else** *S* | **while (** *E* **)** *S* | **{** *L* **}** | *A*

L --> *L* **;** *S* | *S*

Dans un champ *S*.after, on met la première instruction à exécuter après *S*

On initialise *S*.after avant de parcourir *S*

De même pour *L* et *L*.after

Évaluation paresseuse par sauts avec étiquettes symboliques

Nœud **if**

iftrue := newlabel() ;
if.firstchild.true := iftrue ; **if**.firstchild.false := **if**.after ;
visit(**if**.firstchild);
if.secondchild.after=**if**.after;
write(iftrue ':') ; visit(**if**.secondchild) ;

Nœud **while**

begin := newlabel() ; iftrue := newlabel() ;
while.firstchild.true := iftrue ; **while**.firstchild.false := **while**.after ;
write(begin ':') ; visit(**while**.firstchild);
while.secondchild.after := begin;
write(iftrue ':') ; visit(**while**.secondchild) ; write('goto' begin) ;

Évaluation paresseuse par sauts avec étiquettes symboliques

Nœud **ifElse**

iftrue := newlabel() ; iffalse := newlabel() ;
ifElse.firstchild.true := iftrue ; **ifElse**.firstchild.false := iffalse ;
visit(**ifElse**.firstchild);
ifElse.secondchild.after := **ifElse**.after ;
write(iftrue ':') ; visit(**ifElse**.secondchild) ; write('goto' **ifElse**.after) ;
ifElse.thirdchild.after=**ifElse**.after;
write(iffalse ':') ; visit(**ifElse**.thirdchild) ;

Nœud **L**

for each child *S* {
    S.after := newlabel() ; visit(*S*) ; write(*S*.after ':') ; }
write('goto' **L**.after) ;

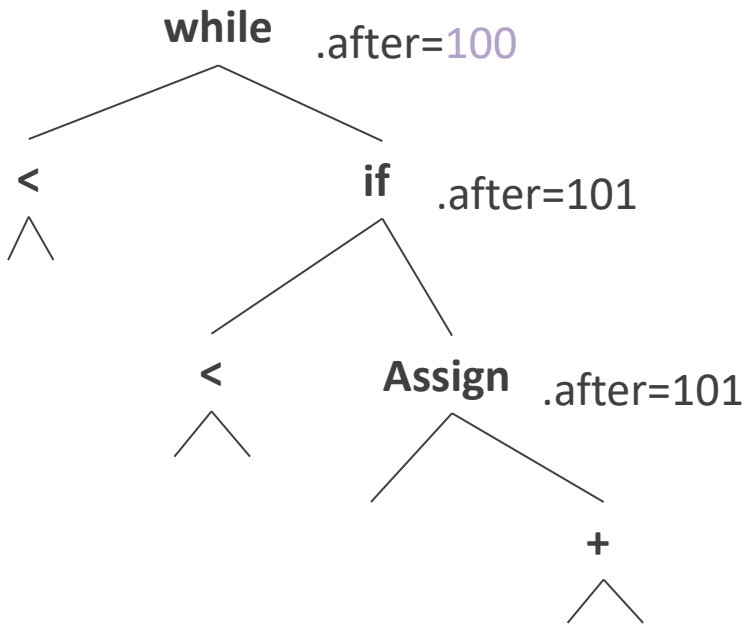
Exemple

Code source

```
while (a < b) if (c < d) x = y + z ;
```

Code intermédiaire

```
101 :  if a < b goto 102
       goto 100
102 :  if c < d goto 103
       goto 101
103 :  t := y + z
       x := t
       goto 101
100 :
```



Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Évaluation paresseuse par sauts avec étiquettes numériques et reprise arrière

S --> **if (** *E* **)** *S* | **if (** *E* **)** *S* **else** *S* | **while (** *E* **)** *S* | **{** *L* **}** | *A*

L --> *L* **;** *S* | *S*

Les sauts vers la première instruction à exécuter après *S* sont engendrés incomplets et listés dans *S*.suivant

De même pour *L* et *L*.suivant

Le compilateur complète ces sauts plus tard

Grammaire attribuée

S --> **if (** *E* **)** *M S*	backpatch(*E*.v, *M*.next) ;
	S.suivant := merge(*E*.f, S_1.suivant) ;
M --> ε	*M*.next := next ;
S --> **while (** *M E* **)** *M S*	backpatch(*E*.v, M_2.next) ;
	backpatch(S_1.suivant, M_1.next) ;
	S.suivant := *E*.f ; write('goto', M_1.next) ;
S --> **if (** *E* **)** *M S* **else** *N S*	backpatch(*E*.v, *M*.next) ;
	backpatch(*E*.f, *N*.next) ;
	S.suivant := merge(S_1.suivant, *N*.suivant, S_2.suivant) ;
N --> ε	*N*.suivant := makeList(next) ; write('goto') ;
	N.next := next ;
S --> **{** *L* **}**	*S*.suivant := *L*.suivant ;
S --> *A*	*S*.suivant := makeList() ;
L --> *L* **;** *M S*	backpatch(L_1.suivant, *M*.next) ; *L*.suivant := *S*.suivant ;
L --> *S*	*L*.suivant := *S*.suivant ;

Exemple

Code source

```
while (a < b) if (c < d) x = y + z ;
```

Code intermédiaire

100 : if a < b goto 102	*E* --> a<b	*E*.v={100}, *E*.f={101}
101 : goto 107	*S* --> while (*M E*) *M*...	M_2.next=102
102 : if c < d goto 104	*E* --> c<d	*E*.v={102}, *E*.f={103}
103 : goto 100	*S* --> if (*E*) *M*...	*M*.next=104
104 : t := y + z	*S* --> if (*E*) *M S*	*S*.suivant={103}
105 : x := t	*S* --> while (*M E*) *M S*	*S*.suivant={101}
106 : goto 100	*S* --> while (*M E*) *M S*	write ('goto', M_1.next)
107 :	*L* --> *L* ; *M*...	*M*.next=107