

LOG3210  
Cours 7

Génération de code intermédiaire (suite)

# Types

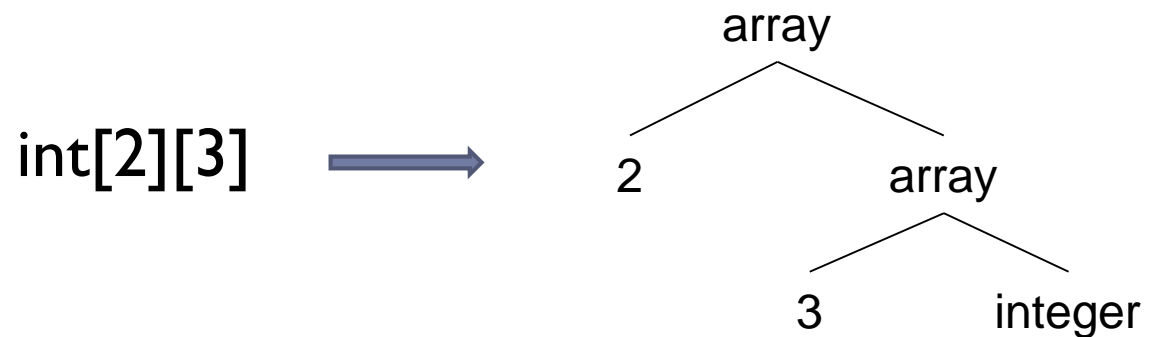
---

- ▶ Les types dans un langage de programmation permettent au compilateur d'effectuer différentes opérations :
  1. Vérifications de types (ex. Les opérandes d'un opérateur logique doivent être des valeurs booléennes.)
  2. Déterminer l'espace requis. (ex. Un entier requiert 4 octets.)
  3. Détecter les conversions de types.
  4. Sélectionner les opérateurs appropriés (ex. + : addition arithmétique ou concaténation de strings)

# Représentation des types

- ▶ Les types peuvent être des structures complexes.
- ▶ Outre les types de base (int, boolean, float, char, etc.) qui sont facilement représentables par attributs, les types plus complexes peuvent requérir une représentation par structures de données spécialisées.

- ▶ Exemple:



# Déclarations

- Nous utiliserons la grammaire suivante pour étudier les déclarations de types:

$D \rightarrow T \text{ id } ; D \mid \varepsilon$

$T \rightarrow B \ C \mid \text{record } \{ \text{ } D \}$

$B \rightarrow \text{int} \mid \text{float}$

$C \rightarrow \varepsilon \mid [ \text{num} ] \ C$

# Déclarations

- En supposant que les **int** occupent 4 bytes, les **float** 8 bytes et en ignorant les **record**, suggérez une SDT pour déterminer le type et l'espace requis pour T. (Fig. 6.15)

$$\begin{array}{ll}
 T \rightarrow B & \{ t = B.type; w = B.width; \} \\
 & C \\
 B \rightarrow \text{int} & \{ B.type = integer; B.width = 4; \} \\
 B \rightarrow \text{float} & \{ B.type = float; B.width = 8; \} \\
 C \rightarrow \epsilon & \{ C.type = t; C.width = w; \} \\
 C \rightarrow [\text{num}] C_1 & \{ array(\text{num.value}, C_1.type); \\
 & \quad C.width = \text{num.value} \times C_1.width; \}
 \end{array}$$

Figure 6.15: Computing types and their widths

# Séquences de déclarations

---

- ▶ Des langages comme le C et le Java permettent de traiter en même temps toutes les déclarations contenues dans une procédure.
- ▶ Au moment d'analyser une procédure, on peut donc considérer que les déclarations sont faites de manière séquentielle et allouer des adresses contigües.
- ▶ Voir Fig. 6.17

# Séquences de déclarations

- Au moment d'analyser une procédure, on peut donc considérer que les déclarations sont faites de manière séquentielle et allouer des adresses contigües.

$$\begin{aligned} P &\rightarrow D \quad \{ \text{offset} = 0; \} \\ D &\rightarrow T \text{ id} ; \quad \{ \text{top.put}(\text{id.lexeme}, T.\text{type}, \text{offset}); \\ &\quad \text{offset} = \text{offset} + T.\text{width}; \} \\ D &\rightarrow \epsilon \end{aligned}$$

Figure 6.17: Computing the relative addresses of declared names

# Champs d'un enregistrement

---

- ▶ Un enregistrement (**record**) dans notre grammaire définit un nouveau *scope* où il est possible de définir des champs qui ont le même nom que d'autres variables précédemment déclarées.

- ▶ Par exemple:

```
...  
float x;  
record { float x; float y; } p;  
record { int tag; float x; int y; } q;
```

- ▶ Voir Fig. 6.18



# Champs d'un enregistrement

- Un enregistrement (**record**) dans notre grammaire définit un nouveau *scope* où il est possible de définir des champs qui ont le même nom que d'autres variables précédemment déclarées.

$$\begin{array}{ll} T \rightarrow \text{record } \{'\} & \{ \text{Env.push(top); top = new Env();} \\ & \text{Stack.push(offset); offset = 0; } \} \\ D \'} & \{ T.type = \text{record(top)}; T.width = \text{offset}; \\ & \text{top = Env.pop(); offset = Stack.pop(); } \} \end{array}$$

Figure 6.18: Handling of field names in records

# Exercice

---

- ▶ À quelle *offset* mémoire se situe la variable `q.x` ?

```
float x;  
record { float x; float y; } p;  
int[42] a;  
record { int tag; float x; int y;} q;
```

- ▶ Dessinez l'arbre de parsage annoté pour ce programme.

# Traduction des expressions

---

- ▶ Une partie importante de la génération de code intermédiaire est la traduction d'expressions en code à 3 adresses.
- ▶ Une expression à plusieurs opérateurs ( $a + b * c$ ) sera traduite en plusieurs instructions à 3 adresses d'une opération chacune.
- ▶ Considérons un exemple simple...

# Traduction des expressions

Production	SDD
$S \rightarrow \text{id} = E ;$	$S.\text{code} = E.\text{code} \parallel$ $\text{gen}(\text{top.get}(\text{id.lexeme}) '=' E.\text{addr})$
$E \rightarrow E_1 + E_2$	$E.\text{addr} = \text{new Temp}()$ $E.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel$ $\text{gen}(E.\text{addr} '=' E_1.\text{addr} + E_2.\text{addr})$
$E \rightarrow - E_1$	$E.\text{addr} = \text{new Temp}()$ $E.\text{code} = E_1.\text{code} \parallel$ $\text{gen}(E.\text{addr} '=' \text{'minus'} E_1.\text{addr})$
$E \rightarrow ( E_1 )$	$E.\text{addr} = E_1.\text{addr}$ $E.\text{code} = E_1.\text{code}$
$E \rightarrow \text{id}$	$E.\text{addr} = \text{top.get}(\text{id.lexeme})$ $E.\text{code} = ''$

- *Top* est une référence à une table de symboles, *Temp()* crée une nouvelle variable temporaire et *gen* génère une instruction à 3 adresses.

# Traduction des expressions

- ▶ As-t-on vraiment besoin de concaténer les instructions générées? Peut-on les générer de manière incrémentielle?

Production	SDT
$S \rightarrow \text{id} = E ;$	{ gen(top.get( <b>id</b> .lexeme) '=' E.addr); }
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp(); gen(E.addr '=' E <sub>1</sub> .addr + E <sub>2</sub> .addr); }
$E \rightarrow - E_1$	{ E.addr = new Temp(); gen(E.addr '=' 'minus' E <sub>1</sub> .addr); }
$E \rightarrow ( E_1 )$	{ E.addr = E <sub>1</sub> .addr; }
$E \rightarrow \text{id}$	{ E.addr = top.get( <b>id</b> .lexeme); }

# Éléments d'un tableau

---

- ▶ En supposant que les index ( $i$ ) d'un tableau commencent à 0, si on connaît le type des éléments et donc l'espace requis pour chaque élément ( $w$ ), il est possible de calculer l'adresse de chacun des éléments à partir de l'adresse de la première case ( $base$ ):

$$base + i \times w$$

- ▶ En plusieurs dimensions:

$$base + i_1 \times w_1 + i_2 \times w_2 + \dots + i_k \times w_k$$

# SDT pour les références aux tableaux

Production	SDT
$S \rightarrow id = E ;$	{ gen(top.get(id.lexeme) '=' E.addr); }
$S \rightarrow L = E ;$	{ gen(L.array.base '[' L.addr ']' '=' E.addr); }
$E \rightarrow E_1 + E_2$	{ E.addr = new Temp(); gen(E.addr '=' E <sub>1</sub> .addr + E <sub>2</sub> .addr); }
$E \rightarrow id$	{ E.addr = top.get(id.lexeme); }
$E \rightarrow L$	{ E.addr = new Temp(); gen(E.addr '=' L.array.base '[' L.addr ']'); }
$L \rightarrow id [ E ]$	{ L.array = top.get(id.lexeme); L.type = L.array.type.elem; L.addr = new Temp(); gen(L.addr '=' E.addr '*' L.type.width); }
$L \rightarrow L_1 [ E ]$	{ L.array = L <sub>1</sub> .array; L.type = L <sub>1</sub> .type.elem; t = new Temp(); L.addr = new Temp(); gen(t '=' E.addr '*' L.type.width); gen(L.addr '=' L <sub>1</sub> .addr '+' t); }

# Vérification de types

---

- ▶ Nous avons vu comment détecter le type des variables au moment de leur déclaration.
- ▶ Au moment de la compilation, le compilateur procède à une vérification des types afin de détecter des erreurs.  
Par exemple:
  - ▶ `int[] array = new char[100];`
  - ▶ `Square sq = new Circle();`
  - ▶ `int[] array = new int[100];`                      `array[0] = 234.67;`



# Typage fort versus faible

---

- ▶ On fera la distinction entre les langages *fortement* et *faiblement* typés.
- ▶ Un compilateur pour un langage fortement typé **garantie** l'absence d'erreurs de types à l'exécution. En d'autres mots, il élimine le besoin de vérifier les types à l'exécution. (D'autres définitions existent.)
- ▶ Si le langage est faiblement typé, le compilateur peut détecter certaines erreurs de types mais ne garantie pas leur absence.

# Synthèse vs. inférence de types

---

- ▶ Lorsqu'il s'agit de vérification de types, le compilateur peut adopter deux stratégies: la synthèse ou l'inférence de types.
- ▶ Si les types des variables sont déclarés avant l'utilisation des variables (C, C++, Java, etc.), il est possible de vérifier les types par synthèse.
- ▶ Exemple:
  - ▶  $c = a + b$  // Cas de base, les types de  $a$ ,  $b$  et  $c$  sont connus
  - ▶  $c = E_1 + E_2$  // On peut synthétiser les types de  $E_1$  et  $E_2$

# Inférence de types

---

- ▶ Si le langage permet l'utilisation de variables dont les types ne sont pas déclarés (Python, PHP, Scala, ML, etc.) il est possible de recourir à l'inférence de types.
- ▶ Exemple en ML, un langage fortement typé :
  - ▶ **fun** length(x) =  
    **if** null(x) **then** 0 **else** length(tl(x)) + 1;
  - ▶ Quelle est le type de la valeur de retour de la fonction length?

# Conversion de types

---

- ▶ Les compilateurs doivent régulièrement convertir le type des variables.
  - ▶ Exemple:  $f + i$  où  $f$  est de type *float* et  $i$  est de type *integer*
- ▶ Il est possible qu'aucune instruction machine ne supporte l'addition de variables de types différents.
- ▶ Dans l'exemple précédent, il faudrait donc convertir  $i$  en un *float*.
- ▶ Augmentons la SDT de l'addition...

# Conversion de types (suite)

Production	SDT
$E \rightarrow E_1 + E_2$	<pre>{ E.addr = new Temp();   if (E<sub>1</sub>.type == integer &amp;&amp; E<sub>2</sub>.type == integer)     E.type = integer;     gen(E.addr '=' E<sub>1</sub>.addr + E<sub>2</sub>.addr);   else {     E.type = float;     tmp = new Temp();     if (E<sub>1</sub>.type == integer) {       gen(tmp '=' '(float)' E<sub>1</sub>.addr)       gen(E.addr '=' tmp + E<sub>2</sub>.addr)     }     else if (E<sub>2</sub>.type == integer) {       gen(tmp '=' '(float)' E<sub>2</sub>.addr)       gen(E.addr '=' E<sub>1</sub>.addr + tmp)     }     else { gen(E.addr '=' E<sub>1</sub>.addr + E<sub>2</sub>.addr); }   } }</pre>

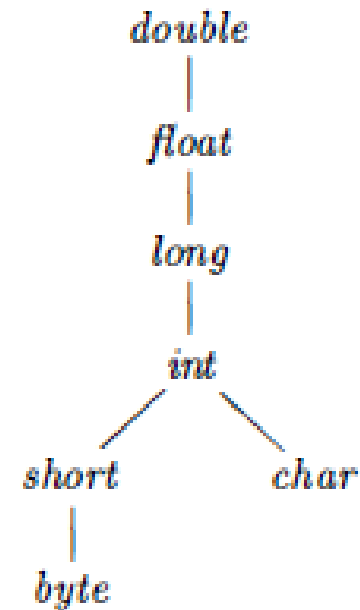
# Conversion de type générale

---

- ▶ Évidemment, à partir du moment où on supporte plus que 2 types, augmenter la SDT de cette façon devient ingérable!
- ▶ Pour traiter le cas général, nous ferons appel à deux fonctions:
  - ▶  $\text{max}(t_1, t_2)$  où  $t_1$  et  $t_2$  sont des types.
  - ▶  $\text{widen}(a, t, w)$  où  $a$  est une adresse et où  $t$  et  $w$  sont des types.

# Fonction $\max(t_1, t_2)$

- ▶ La fonction  $\max(t_1, t_2)$  prend deux types en paramètre et retourne le premier type en mesure de représenter  $t_1$  et  $t_2$  sans perte d'information.
- ▶ Dans l'arbre des conversions à droite, la valeur de  $\max(t_1, t_2)$  correspond au plus petit ancêtre commun de  $t_1$  et  $t_2$ .
- ▶ Max retourne une erreur s'il reçoit un paramètre qui n'est pas un type valide.



# Fonction `widen(a, t, w)`

- ▶ Cette fonction génère les instructions de code intermédiaire requises pour convertir une variable de type  $t$  en un type  $\geq w$ .
- ▶ En d'autres mots,  $w$  doit être égal à  $t$  ou être un parent de  $t$  dans l'arbre des conversions.
- ▶ Revisions la SDT des expressions...

```
Addr widen(Addr a, Type t, Type w)
    if ( t = w ) return a;
    else if ( t = integer and w = float ) {
        temp = new Temp();
        gen(temp '=' '(float)' a);
        return temp;
    }
    else error;
}
```

Figure 6.26: Pseudocode for function *widen*



# Conversion de types générique

- Les fonctions `max` et `widen` permettent d'isoler la logique de conversion de types de la SDT.

Production	SDT
$E \rightarrow E_1 + E_2$	<pre>{ E.addr = new Temp();   E.type = max(E<sub>1</sub>.type, E<sub>2</sub>.type);   a<sub>1</sub> = widen(E<sub>1</sub>.addr, E<sub>1</sub>.type, E.type);   a<sub>2</sub> = widen(E<sub>2</sub>.addr, E<sub>2</sub>.type, E.type);   gen(E.addr '=' a<sub>1</sub> '+' a<sub>2</sub>); }</pre>

- Exercice 6.5.1

# Surcharge de fonctions et d'opérateurs

- ▶ La surcharge de fonctions et d'opérateurs est commune dans les langages de programmation. Les types permettront de déterminer quelle version de la fonction/opérateur doit être appelée.
- ▶ Dans les langages comme C, C++ et Java, la surcharge est résolue en fonction des types des arguments de la fonction.
- ▶ En d'autres mots, la surcharge est résolue en faisant correspondre les signatures des fonction appelée et déclarée.

# Exemple - Surcharge

---

```
#include <iostream>
using namespace std;
void print(int i) {
    cout << " Here is int " << i << endl;
}
void print(double f) {
    cout << " Here is double " << f << endl;
}
void print(string c) {
    cout << " Here is string " << c << endl;
}

int main() {
    print(10);          //print(10) ⇔ print(int i)
    print(10.10);       //print(10.10) ⇔ print(double f)
    print("ten");        //print("ten") ⇔ print(string c)
}
```

# Exemple – Surcharge (2)

---

```
#include <iostream>
using namespace std;
void print(double f) {
    cout << " Here is double " << f << endl;
}
void print(string c) {
    cout << " Here is string " << c << endl;
}

int main() {
    print(10);          //print(10) ⇔ print(double f)
    print(10.10);      //print(10.10) ⇔ print(double f)
    print("ten");      //print("ten") ⇔ print(string c)
}
```

# Exemple – Surcharge (Bonus!)

---

```
#include <iostream>
using namespace std;

void print(int i, double j) {
    cout << " Here is int " << i << endl;
    cout << " Here is double " << j << endl;
}

void print(double i, int j) {
    cout << " Here is double " << i << endl;
    cout << " Here is int " << j << endl;
}

int main() {
    print(10,10);
}
```

Retourne quoi...?

# Flux de contrôle

---

- ▶ Les structures de contrôle induisent des branchements qui sont résolus à l'exécution.
- ▶ Par exemple, les **if**, **while**, **for**, **switch**, etc. sont toutes des structures de contrôle.
- ▶ Traduire des structures de contrôle en code intermédiaire requière évidemment d'être en mesure de traduire des expressions Booléennes.

# Expressions Booléennes

---

- ▶ Une expression Booléenne est simplement une expression qui retourne vrai ou faux. La grammaire suivante définit les expressions Booléennes que nous supporterons:
- ▶  $B \rightarrow B \parallel B \mid B \&\& B \mid !B \mid ( B ) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$
- ▶ où **rel** est un opérateur relationnel de la forme  $<$ ,  $<=$ ,  $=$ ,  $!=$ ,  $>$ , ou  $>=$ .

# Court-circuit

---

- ▶ Étant donné une expression Booléenne, il est parfois possible de connaître le résultat de l'expression après une évaluation partielle.
- ▶ Par exemple: `if ( x < 100 || x > 200 && x != y) x = 0;`
  - ▶ Si  $x < 100$ , l'expression est vraie
  - ▶ Si  $x \geq 100$  et que  $x \leq 200$ , l'expression est fausse
  - ▶ Si  $x \geq 100$  et  $x > 200$ , alors il faut vérifier si  $x \neq y$
- ▶ À quoi ressemblerait le code intermédiaire?



# Traduction des structures de contrôle

- ▶ Nous construirons tout d'abord une SDT pour les structures de contrôle qui répondent à la grammaire suivante:

$$S \rightarrow \text{if} ( B ) S_1$$
$$S \rightarrow \text{if} ( B ) S_1 \text{ else } S_2$$
$$S \rightarrow \text{while} ( B ) S_1$$

- ▶ où  $B$  est un non terminal qui représente une expression Booléenne.

# Traduction des structures de contrôle (suite)

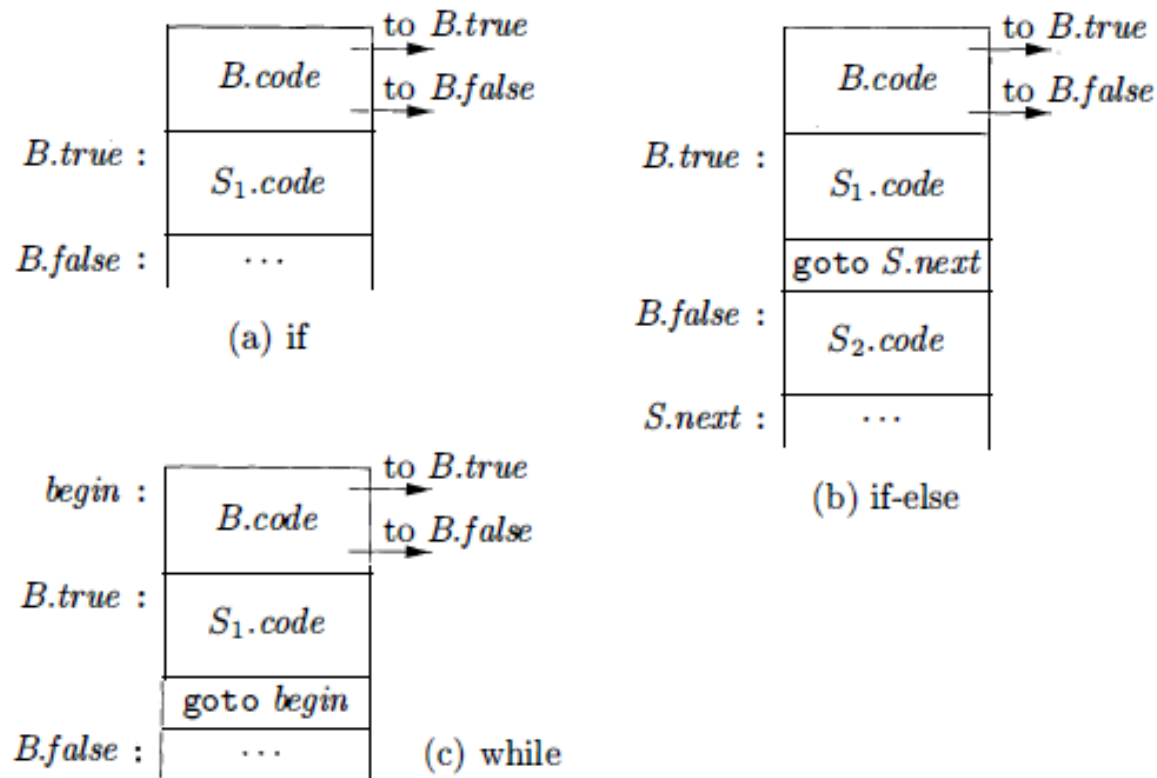


Figure 6.35: Code for if-, if-else-, and while-statements

# SDD – Structures de contrôle

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign}.code$
$S \rightarrow \text{if} ( B ) S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel label(B.true) \parallel S_1.code$
$S \rightarrow \text{if} ( B ) S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' S.next)$ $\parallel label(B.false) \parallel S_2.code$
$S \rightarrow \text{while} ( B ) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

# SDD – Expressions Booléennes

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \    \ B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \    \ label(B_1.false) \    \ B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \    \ label(B_1.true) \    \ B_2.code$
$B \rightarrow ! \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ rel \ E_2$	$B.code = E_1.code \    \ E_2.code$ $\quad \    \ gen('if' \ E_1.addr \ rel.op \ E_2.addr \ 'goto' \ B.true)$ $\quad \    \ gen('goto' \ B.false)$
$B \rightarrow true$	$B.code = gen('goto' \ B.true)$
$B \rightarrow false$	$B.code = gen('goto' \ B.false)$

# Exemple de traduction

---

Traduisons le segment de code suivant avec les SDD précédentes:

```
if ( x < 100 || x > 200 && x != y) x = 0;
```

Le résultat est-il optimal?

# Élimination des GOTO redondants

- ▶ Dans l'exemple précédent, nous générions le code intermédiaire suivant:

```
        if x > 200 goto L4
        goto L1
L4:    ...
```

- ▶ Il est simple de constater que l'instruction suivante accompli la même tâche:

```
        ifFalse x > 200 goto L1
L4:    ...
```

# Élimination des GOTO redondants (suite)

- L'idée est de profiter de la séquence naturelle des instructions et d'éviter des GOTO inutiles.
- L'instruction *fall* servira à signaler qu'il ne faut pas générer de GOTO.
- Les instructions de type **if-else** et **while** fixent aussi *B.true* à *fall*.

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \parallel label(S.next)$
$S \rightarrow \text{assign}$	$S.code = \text{assign.code}$
$S \rightarrow \text{if} ( B ) S_1$	$B.true = fall$ $B.false = S_1.next = S.next$ $S.code = B.code \parallel S_1.code$
$S \rightarrow \text{if} ( B ) S_1 \text{ else } S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' S.next)$ $\parallel label(B.false) \parallel S_2.code$
$S \rightarrow \text{while} ( B ) S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \parallel B.code$ $\parallel label(B.true) \parallel S_1.code$ $\parallel gen('goto' begin)$
$S \rightarrow S_1 S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \parallel label(S_1.next) \parallel S_2.code$

# Traduction des opérateurs relationnels

```
test = E1.addr rel.op E2.addr  
  
s = if B.true ≠ fall and B.false ≠ fall then  
    gen('if' test 'goto' B.true) || gen('goto' B.false)  
    else if B.true ≠ fall then gen('if' test 'goto' B.true)  
    else if B.false ≠ fall then gen('ifFalse' test 'goto' B.false)  
    else ''  
  
B.code = E1.code || E2.code || s
```

Figure 6.39: Semantic rules for  $B \rightarrow E_1 \text{ rel } E_2$



# Traduction des opérateurs logiques

- Dans ce cas-ci, si  $B.true == fall$ , il faut tout de même s'assurer de ne pas calculer  $B_2$  inutilement. Si  $B.true == fall$ ,  $B_1$  va générer un GOTO pour passer par-dessus les instructions de  $B_2$ .

```
 $B_1.true = \text{if } B.true \neq fall \text{ then } B.true \text{ else } newlabel()$   
 $B_1.false = fall$   
 $B_2.true = B.true$   
 $B_2.false = B.false$   
 $B.code = \text{if } B.true \neq fall \text{ then } B_1.code \parallel B_2.code$   
           $\text{else } B_1.code \parallel B_2.code \parallel label(B_1.true)$ 
```

Figure 6.40: Semantic rules for  $B \rightarrow B_1 \parallel B_2$

# Expressions Booléennes hors des structures de contrôle

- ▶ Les expressions Booléennes peuvent être utilisées en dehors des structures de contrôle.
- ▶ Exemple:  $x = a < b \ \&\& \ c < d$
- ▶ Il faudra alors générer le même code avec les GOTO, mais s'assurer que le résultat est bien stocké dans une variable temporaire!
- ▶ Une stratégie consiste à faire 2 traversements de l'arbre de parsage. Une fois pour les expressions dans les structures de contrôles et une fois pour les autres expressions en variant la SDT.

# Traduction des switch

- Normalement, la manière la plus efficace d'implanter un switch est d'utiliser un tableau (< 10 éléments) ou une table de hachage (>= 10 éléments).

```
switch (age) {
    case 1: S1
    case 2: S2
    ...
    case 99: Sn-1
    default: Sn
}
```

```
L1: S1
L2: S2
...
Ln-1: Sn-1
Ln: Sn
```

Valeur	Étiquette
1	L <sub>1</sub>
2	L <sub>2</sub>
...	...
99	L <sub>n-1</sub>
default	L <sub>n</sub>

# Traduction des switch par SDT

```
switch ( E ) {
    case  $V_1$ :  $S_1$ 
    case  $V_2$ :  $S_2$ 
        ...
    case  $V_{n-1}$ :  $S_{n-1}$ 
    default:  $S_n$ 
}
```

Figure 6.48: Switch-statement syntax

```

                                code to evaluate  $E$  into  $t$ 
                                goto test
L1:                            code for  $S_1$ 
                                goto next
L2:                            code for  $S_2$ 
                                goto next
                                ...
L $n-1$ :                        code for  $S_{n-1}$ 
                                goto next
L $n$ :                          code for  $S_n$ 
                                goto next
test:                          if  $t = V_1$  goto L1
                                if  $t = V_2$  goto L2
                                ...
                                if  $t = V_{n-1}$  goto L $n-1$ 
                                goto L $n$ 
next:
```

Figure 6.49: Translation of a switch-statement


# Traduction des switch par SDT (suite)

1. Quand le mot clé **switch** est rencontré, générer deux étiquettes: *test* et *next* et un temporaire *t*.
2. Générer le code pour évaluer *E* et stocker le résultat dans *t*.
3. Générer un goto vers *test*
4. Pour chaque mot clé **case**, créer une étiquette  $L_i$  et l'insérer dans la table de symboles.
5. Insérer une paire  $(V_i, L_i)$  représentant la constante du **case** et son étiquette dans une file.
6. Émettre l'étiquette  $L_i$  accompagnée du code  $S_i$  suivi d'un goto *next*.
7. Quand il n'y a plus de **case**, lire la file et émettre une instruction « **case**  $t V_i L_i$  »

# Traduction des switch par SDT (suite)

```
        code to evaluate  $E$  into  $t$ 
        goto test
L1:    code for  $S_1$ 
        goto next
L2:    code for  $S_2$ 
        goto next
        ...
L $n-1$ : code for  $S_{n-1}$ 
        goto next
L $n$ :   code for  $S_n$ 
        goto next
test:   if  $t = V_1$  goto L1
        if  $t = V_2$  goto L2
        ...
        if  $t = V_{n-1}$  goto L $n-1$ 
        goto L $n$ 
next:
```

Cette forme est plus facile à convertir en table de hachage par le générateur de code.



```
test:   case  $t V_1$  L1
        case  $t V_2$  L2
        ...
        case  $t V_{n-1}$  L $n-1$ 
        case  $t t$  L $n$ 
next:
```

Figure 6.49: Translation of a switch-statement