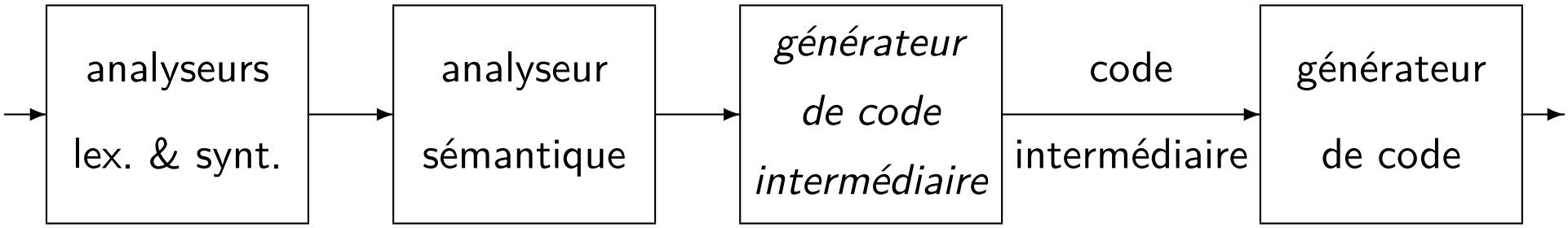
Génération de code intermédiaire

Sections 6.1 à 6.6 et 6.8

* Contenu *

- Représentation intermédiaire
  - Code à trois adresses
- Traduction de code
  - Gestion des tableaux
  - Traduction des énoncés de contrôle
    - Sans court-circuit
    - Avec court-circuit
  - Traduction des expressions booléennes
    - Sans court-circuit
    - Avec court-circuit
- Vérification de la sémantique
  - Vérification de types
  - Gestion des déclarations

Introduction



La génération de code intermédiaire peut être considérée comme le point où on passe de la partie analyse à la partie synthèse.

En principe, il reste peu ou pas de traces du langage source à ce point. Aussi, il y apparaît peu ou pas de traces du langage (ou de la machine) cible.

Bien qu'optionnel, le passage par le code intermédiaire apporte certains avantages:

- il est relativement aisé de changer le langage cible du compilateur;
- la représentation intermédiaire permet d'appliquer des optimisations indépendantes du langage cible.

Représentations intermédiaires

Section 6.1

On connaît déjà les représentations intermédiaires suivantes:

- les arbres de syntaxe (avec ou sans partage des sous-expressions communes);
- la notation postfixe.

Exemple: représentations de '`a := (b + c) * (b + c) * (- d)`'.

On introduit maintenant la représentation du code à trois adresses:

- il s'agit de séquences d'énoncés simples dont les plus courants sont de la forme '`x := y` *op* `z`' (2 opérandes + 1 receveur = 3 adresses);
- le découpage des expressions complexes du langage source en des séquences d'énoncés simples demande l'introduction de variables temporaires $\mathtt{t}_i$.

Code à trois adresses

Section 6.2

Voici les types d'énoncés que l'on retrouve dans le code à trois adresses:

- '`x := y` *op* `z`' où *op* est un opérateur binaire;
- '`x :=` *op* `y`' où *op* est un opérateur unaire;
- '`x := y`';
- '`goto L`', où `L` est l'étiquette d'un énoncé dans la séquence;
- '`if x` *relop* `y goto L`' où *relop* est un opérateur de comparaison comme $\leq$, $>$, $\neq$, etc.;
- '`x := y[i]`' et '`x[i] := y`' pour les accès dans des tableaux;
- '`x := &y`', '`x := *y`' et '`*x := y`' pour la manipulation des pointeurs;
- '`L :`', pseudo-énoncé qui déclare l'étiquette `L`, il marque la position de l'énoncé suivant.

Code à trois adresses

Le choix du jeu d'instructions de la représentation intermédiaire doit être fait judicieusement.

- Il doit être assez riche pour pouvoir exprimer tous les calculs possibles dans le langage source.
- S'il est trop étendu, le générateur de code devient alors lourd; tout changement de langage cible est coûteux.
- S'il est trop restreint, les séquences de code intermédiaire à générer deviennent plus longues; la réalisation des optimisations subséquentes est plus difficile.

Implantation des énoncés à 3 adresses

Pour le cours, nous nous en tenons à l'implantation par *quadruplets*.

- Permet plus de souplesse dans la réalisation d'optimisations.
- Les économies d'espace liées aux implantations par *triplets* ou *triplets indirects* ne valent plus tellement la peine de nos jours.

Exemple:

	op	$\mathbf{arg_1}$	$\mathbf{arg_2}$	*résultat*
0	`minus`	`c`		t_1
1	`*`	`b`	t_1	t_2
2	`minus`	`c`		t_3
3	`*`	`b`	t_3	t_4
4	`+`	t_2	t_4	t_5
5	`=`	t_5		`a`

Génération de code:

Traduction du code

Traduction des expressions

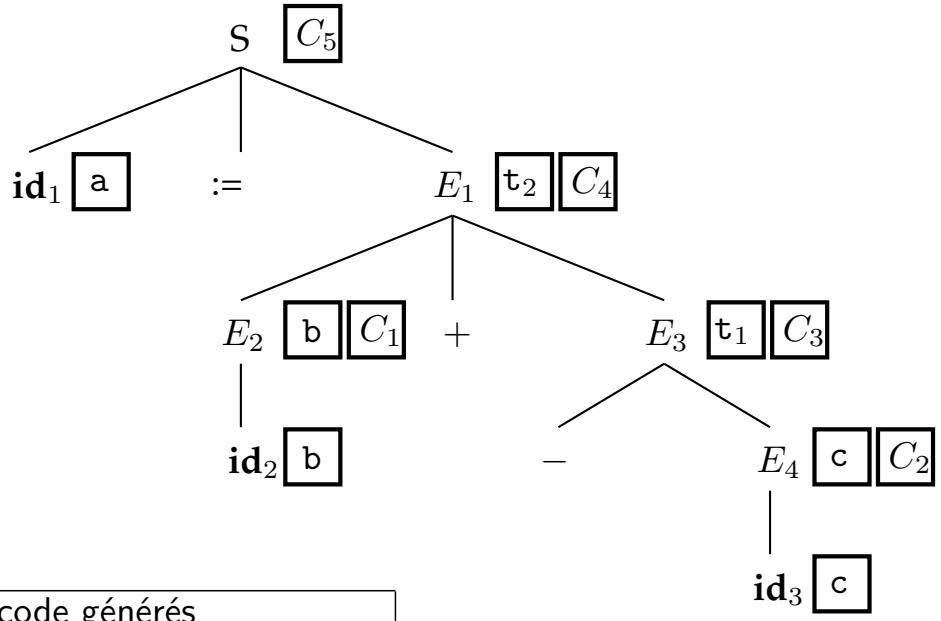
Section 6.4

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

Le symbole S synthétise l'attribut $code$. Le symbole E synthétise les attributs $place$ et $code$.

Traduction des expressions

Exemple: Génération de code intermédiaire pour '`a := b + -c`'.



Morceaux de code générés		
	En bref	Au long
C_1	—	—
C_2	—	—
C_3	C_2 t_1 := uminus c	t_1 := uminus c
C_4	C_1 C_3 t_2 := b + t_1	t_1 := uminus c t_2 := b + t_1
C_5	C_4 a := t_2	t_1 := uminus c t_2 := b + t_1 a := t_2

$$C_5\left\{\begin{array}{l} C_4\left\{\begin{array}{l} C_3\left\{\begin{array}{l} C_1\left\{\begin{array}{l} \text{—} \end{array}\right. \\ C_2\left\{\begin{array}{l} \text{—} \end{array}\right. \\ \texttt{t}_1 := \texttt{uminus c} \\ \texttt{t}_2 := \texttt{b} + \texttt{t}_1 \end{array}\right. \\ \texttt{a} := \texttt{t}_2 \end{array}\right. \end{array}\right.$$

Tableaux

Cette section pousse plus loin la génération de code pour divers énoncés. Nous ne faisons qu'éclaircir la question de l'accès aux tableaux.

Considérons l'accès à un tableau unidimensionnel '`A[i]`' où '`A`' est stocké à l'adresse $base$ en mémoire, où l'index peut aller de low à $high$ et où les éléments de base du tableau sont de taille w. Alors, l'adresse de '`A[i]`' est (équation 8.6):

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

On peut améliorer l'efficacité de l'expression en la réécrivant comme:

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

Exemple: organisation en mémoire de `A[`i`]`, où on fixe $low = 2$ et $high = 6$:

Tableaux

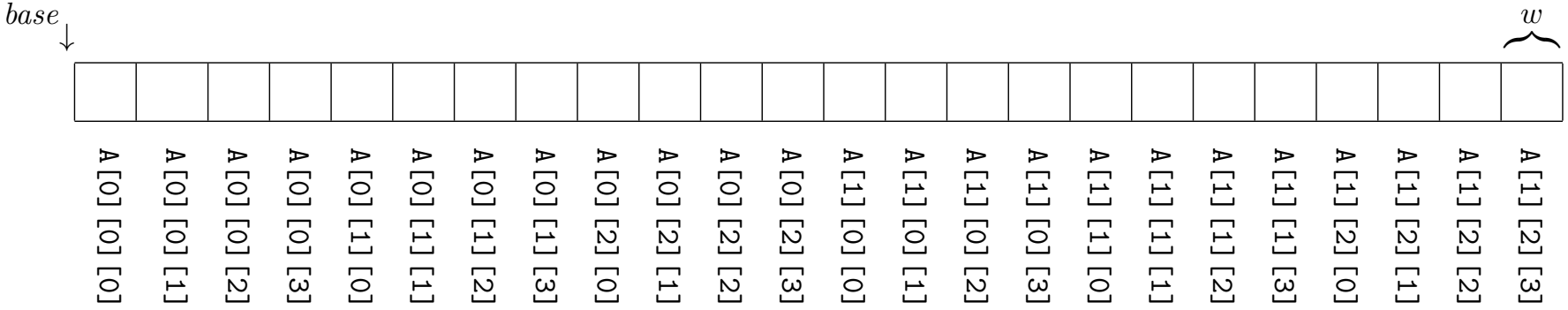
Si on considère maintenant un accès à un tableau multidimensionnel '`A[i`$_1$`]...[i`$_k$`]`' où l'index de la $i$ème dimension peut aller de low_i à $high_i$, i.e. pouvant prendre $n_i = high_i - low_i + 1$ valeurs différentes, l'adresse de '`A[i`$_1$`]...[i`$_k$`]`' devient alors:

$$base + ((\ldots(((i_1 - low_1) \times n_2 + i_2 - low_2) \times n_3 + i_3 - low_3)\ldots) \times n_k + i_k - low_k) \times w$$

et on peut améliorer l'efficacité de cette formule en la réécrivant ainsi:

$$\begin{gathered}((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3)\ldots) \times n_k + i_k) \times w \\ + base - ((\ldots((low_1 \times n_2 + low_2) \times n_3 + low_3)\ldots) \times n_k + low_k) \times w\end{gathered}$$

Exemple: organisation en mémoire de `A[`i_1`][`i_2`][`i_3`]`, où on fixe $low_1 = low_2 = low_3 = 0$, $high_1 = 1$, $high_2 = 2$ et $high_3 = 3$:



Traduction des énoncés de contrôle

Sections 6.6 et 6.8

Soit le code `if (x > 7) then S1 else S2`, étant donnée la production $S \rightarrow$ **if** E **then** S_1 **else** S_2. Il existe deux techniques de génération de code pour gérer les sauts conditionnels:

- le code "classique";
- le code à "court circuit".

Exemple:

```
t1 := x > 7
if (t1 = 0) goto Lelse
S1.code
goto Lfin
Lelse:
S2.code
Lfin:
```

(a) Code "classique"

```
if (x <= 7) goto Lelse
S1.code
goto Lfin
Lelse:
S2.code
Lfin:
```

(b) Code à "court circuit"

Traduction des énoncés de contrôle

Les deux techniques de génération de code ont les caractéristiques suivantes:

- **Code "classique"**: on génère le code pour la condition; le résultat de la comparaison est placée dans une variable. Par la suite, on lit le contenu de la variable afin d'effectuer un saut conditionnel.
  - E génère du code calculant l'expression booléenne, puis synthétise l'adresse de la variable contenant le résultat. S génère alors du code qui effectue le saut selon la valeur obtenue par E.
- **Code à "court circuit"**: la comparaison est utilisée immédiatement dans l'instruction de saut conditionnel.
  - Ce style de code s'exécute souvent un plus rapidement que le code "traditionnel" mais est un peu plus complexe à générer.
  - E reçoit deux attributs hérités: une étiquette qui indique où brancher si la condition est vraie, et une étiquette qui indique où brancher si la condition est fausse.

Traduction des énoncés de contrôle

Quelques conventions à propos du langage de programmation hypothétique que nous traduisons.

- Une boucle '**while**' teste la condition avant toute itération; elle pourrait ne faire aucune itération.
- Une boucle '**do ... while**' teste la condition après toute itération; elle doit faire au moins une itération.
- Un branchement '**switch**' n'exécute qu'une seule branche, soit la branche sélectionnée par la clé; c'est le contraire de ce qui se passe en C, où une branche doit se terminer par '**break**', sans quoi le contrôle tombe dans la branche suivante.

Traduction des énoncés de contrôle

Quelques conventions (suite. . .)

- La valeur booléenne *fausse* est représentée par zéro (0) et la valeur booléenne *vraie* est représentée par tout autre nombre.

- Les opérateurs logiques '**and**' et '**or**' sont les "et" et "ou" *intelligents*.

  C'est-à-dire qu'ils mettent fin à leur calcul dès que la réponse est connue, logiquement parlant.

  Lorsque l'un des opérateurs doit produire "vrai", la valeur produite doit être égale à la valeur de la sous-expression qui rend l'expression vraie; i.e.:

  - si E_1 **and** E_2 produit "vrai", la valeur produite est égale à celle de E_2;
  - si E_1 **or** E_2 produit "vrai", la valeur produite est égale à celle de E_1, si celle-ci est vraie, et égale à celle de E_2, autrement.

Traduction classique des énoncés de contrôle: if

Allure du code généré:

```
E.code
if E.place = 0 goto Lelse
S1.code
goto Lsortie

Lelse:
S2.code

Lsortie:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$S.L_{else} :=$ **new** Label $S.L_{sortie} :=$ **new** Label $S.code := E.code \parallel$ $gen(\text{`}\textbf{if}\text{'}\ E.place\ \text{`}=\text{'}\ \text{`}0\text{'}\ \text{`}\textbf{goto}\text{'}\ S.L_{else}) \parallel$ $S_1.code \parallel gen(\text{`}\textbf{goto}\text{'} S.L_{sortie}) \parallel$ $gen(S.L_{else}\ \text{`}:\text{'}) \parallel S_2.code \parallel gen(S.L_{sortie}\ \text{`}:\text{'})$

Traduction classique des énoncés de contrôle: do . . . while

Allure du code généré:

```
Lbegin:
S1.code
E.code
if E.place ≠ 0 goto Lbegin
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$S \rightarrow$ **do** S_1 **while** E	$S.begin :=$ **new** Label $S.code := gen(S.begin\text{`:'}) \parallel$ $S_1.code \parallel E.code \parallel$ $gen(\text{`}$**if**$\text{'}\ E.place\ \text{`}\neq\text{'}\ \text{`0'}\ \text{`}$**goto**$\text{'}\ S.begin)$

Traduction classique des énoncés de contrôle: while

Allure du code généré:

```
Lbegin:
E.code
if E.place = 0 goto Lafter
S1.code
goto Lbegin
Lafter:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$S \rightarrow$ **while** E **do** S_1	$S.begin :=$ **new** Label $S.after :=$ **new** Label $S.code := gen(S.begin\text{`:'}) \parallel$ $E.code \parallel$ $gen(\text{`}$**if**$\text{'}\ E.place\ \text{`='}\ \text{`0'}\ \text{`}$**goto**$\text{'}\ S.after) \parallel$ $S_1.code \parallel$ $gen(\text{`}$**goto**$\text{'}\ S.begin) \parallel$ $gen(S.after\ \text{`:'})$

Note: Il y a moyen de générer du code plus efficace pour le **while** à raison d'un saut par itération, au lieu de deux.

Traduction classique des énoncés de contrôle: switch

Nous utiliserons cette grammaire:

$$
\begin{array}{rcll}
S & \rightarrow & \textbf{switch } E \textbf{ with } C & \\
C & \rightarrow & \textbf{case num} : S;\ C_1 & \\
 & | & \textbf{else } S & \texttt{/* clause } \textbf{else} \texttt{ obligatoire */}
\end{array}
$$

Exemple de code utilisant cette grammaire:

```
switch E with
    case 12 : S1;
    case 14 : S2;
    else S3
```

Allure du code généré:

```
E.code

if E.place ≠ 12 goto Lfin12
S1.code
goto Lsortie
Lfin12:

if E.place ≠ 14 goto Lfin14
S2.code
goto Lsortie
Lfin14:

S3.code
Lsortie:
```

Traduction classique des énoncés de contrôle: switch

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$S \rightarrow$ **switch** E **with** C	$C.sortie :=$ **new** Label $C.temp := E.place$ $S.code := E.code \parallel C.code$
$C \rightarrow$ **case num** : $S; C_1$	$C.L_1 :=$ **new** Label $C_1.temp := C.temp$ $C_1.sortie := C.sortie$ $C.code := gen(\text{`}\textbf{if}\text{'}\ C.temp\ \text{`}\neq\text{'}\ \textbf{num}.val\ \text{`}\textbf{goto}\text{'}\ C.L_1) \parallel$ $S.code \parallel$ $gen(\text{`}\textbf{goto}\text{'}\ C.sortie) \parallel$ $gen(C.L_1\text{`:'}) \parallel$ $C_1.code$
$C \rightarrow$ **else** S	$C.code := S.code \parallel gen(C.sortie\text{`:'})$

Traduction des énoncés de contrôle avec court-circuit: if

Exemple de code source:

```
if E1 > E2 then
    S1
else
    S2
```

Allure du code généré:

```
        E1.code
        E2.code
        if E1.place > E2.place goto Lthen
        goto Lelse
    Lthen:
        S1.code
        goto Lfin
    Lelse:
        S2.code
    Lfin:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$S \rightarrow$ **if** B **then** S_1 **else** S_2	$S.L_{fin} :=$ **new** Label $B.L_{true} :=$ **new** Label $B.L_{false} :=$ **new** Label $S.code := B.ccode \parallel gen(B.L_{true}\ \text{':'}) \parallel$ $S_1.code \parallel gen(\text{'}\textbf{goto}\text{'}\ S.L_{fin}) \parallel$ $gen(B.L_{false}\ \text{':'}) \parallel S_2.code \parallel gen(S.L_{fin}\ \text{':'})$
$B \rightarrow E_1 > E_2$	$B.ccode := E_1.code \parallel E_2.code \parallel$ $gen(\text{'}\textbf{if}\text{'}\ E_1.place\ \text{'>'}\ E_2.place\ \text{'}\textbf{goto}\text{'}\ B.L_{true}) \parallel$ $gen(\text{'}\textbf{goto}\text{'} B.L_{false})$

Traduction des énoncés de contrôle avec court-circuit: do . . . while

Exemple de code source:

```
do
    S1
while E1 > E2
```

Allure du code généré:

```
Ltrue:
    S1.code
    E1.code
    E2.code
    if E1.place > E2.place goto Ltrue
    goto Lfalse
Lfalse:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$S \rightarrow$ **do** S_1 **while** B	$B.L_{true} :=$ **new** Label $B.L_{false} :=$ **new** Label $S.code := gen(B.L_{true}$ ':') $\parallel S_1.code \parallel$ $B.ccode \parallel gen(B.L_{false}$ ':')
$B \rightarrow E_1 > E_2$	$B.ccode := E_1.code \parallel E_2.code \parallel$ $gen($'**if**' $E_1.place$ '$>$' $E_2.place$ '**goto**' $B.L_{true}) \parallel$ $gen($'**goto**' $B.L_{false})$

Traduction des énoncés de contrôle avec court-circuit: while

Exemple de code source:

```
while E1 > E2 do
    S1
```

Allure du code généré:

```
Lbegin:
    E1.code
    E2.code
    if E1.place > E2.place goto Ltrue
    goto Lfalse
Ltrue:
    S1.code
    goto Lbegin
Lfalse:
```

Définition orientée-syntaxe:

Production	Règles sémantiques
$S \rightarrow$ **while** B **do** S_1	$S.L_{begin} :=$ **new** Label
	$B.L_{true} :=$ **new** Label
	$B.L_{false} :=$ **new** Label
	$S.code := gen(S.L_{begin}$ ':') $\parallel B.ccode \parallel$
	$gen(B.L_{true}$ ':') $\parallel S_1.code \parallel$
	$gen($'**goto**' $S.L_{begin}) \parallel gen(B.L_{false}$ ':')
$B \rightarrow E_1 > E_2$	$B.ccode := E_1.code \parallel E_2.code \parallel$
	$gen($'**if**' $E_1.place$ '>' $E_2.place$ '**goto**' $B.L_{true}) \parallel$
	$gen($'**goto**' $B.L_{false})$

Note: Il y a moyen de générer du code plus efficace pour le **while** à raison d'un saut par itération, au lieu de deux.

Traduction classique des expressions booléennes: E_1 and E_2

Exemple de code source:

```
/* E: */
E1 and E2
```

Allure du code généré:

```
E1.code
if E1.place ≠ 0 goto Ltrue
E.place := 0
goto Lend

Ltrue:
E2.code
E.place := E2.place

Lend:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$E \rightarrow E_1$ **and** E_2	$E.place :=$ **new** Temp $E.L_{true} :=$ **new** Label $E.L_{end} :=$ **new** Label $E.code := E_1.code \parallel$ $gen(\text{`}\textbf{if}\text{'}\ E_1.place\ \text{`}\neq\text{'}\ \text{`}0\text{'}\ \text{`}\textbf{goto}\text{'}\ E.L_{true}) \parallel$ $gen(E.place\ \text{`}:=\text{'}\ \text{`}0\text{'}) \parallel gen(\text{`}\textbf{goto}\text{'}\ E.L_{end}) \parallel$ $gen(E.L_{true}\ \text{`}:\text{'}) \parallel E_2.code \parallel$ $gen(E.place\ \text{`}:=\text{'}\ E_2.place) \parallel gen(E.L_{end}\ \text{`}:\text{'})$

Traduction classique des expressions booléennes: E_1 **or** E_2

Exemple de code source:

```
/* E: */
E1 or E2
```

Allure du code généré:

```
E1.code
if E1.place = 0 goto Lfalse
E.place := E1.place
goto Lend

Lfalse:
E2.code
E.place := E2.place

Lend:
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$E \rightarrow E_1$ **or** E_2	$E.place :=$ **new** Temp $E.L_{false} :=$ **new** Label $E.L_{end} :=$ **new** Label $E.code := E_1.code \parallel$ $gen(\text{`}\textbf{if}\text{'}\ E_1.place\ \text{`}=\text{'}\ \text{`}0\text{'}\ \text{`}\textbf{goto}\text{'}\ E.L_{false}) \parallel$ $gen(E.place\ \text{`}:=\text{'}\ E_1.place) \parallel gen(\text{`}\textbf{goto}\text{'}\ E.L_{end}) \parallel$ $gen(E.L_{false}\ \text{`}:\text{'}) \parallel E_2.code \parallel$ $gen(E.place\ \text{`}:=\text{'}\ E_2.place) \parallel gen(E.L_{end}\ \text{`}:\text{'})$

Traduction d'expressions avec court-circuit: B_1 and B_2

Exemple de code source:

```
if (B1 and B2) then
    ...
else
    ...
```

Allure du code généré:

```
    B1.ccode // branche à Lmiddle ou Lfalse
    Lmiddle:
    B2.ccode // branche à Ltrue ou Lfalse

    Ltrue:
        ...
    Lfalse:
        ...
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$B \rightarrow B_1$ **and** B_2	$B_1.L_{true} :=$ **new** Label $B_1.L_{false} := B.L_{false}$ $B_2.L_{true} := B.L_{true}$ $B_2.L_{false} := B.L_{false}$ $B.ccode := B_1.ccode \parallel gen(B_1.L_{true}\ \text{':'}) \parallel B_2.ccode$

Traduction d'expressions avec court-circuit: B_1 **or** B_2

Exemple de code source:

```
if (B1 or B2) then
    ...
else
    ...
```

Allure du code généré:

```
    B1.ccode // branche à Ltrue ou Lmiddle
    Lmiddle:
    B2.ccode // branche à Ltrue ou Lfalse

    Ltrue:
        ...
    Lfalse:
        ...
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$B \rightarrow B_1$ **or** B_2	$B_1.L_{true} := B.L_{true}$ $B_1.L_{false} :=$ **new** Label $B_2.L_{true} := B.L_{true}$ $B_2.L_{false} := B.L_{false}$ $B.ccode := B_1.ccode \parallel gen(B_1.L_{false}\ \text{':'}) \parallel B_2.ccode$

Traduction d'expressions avec court-circuit: not B_1

Pour effectuer la négation, il suffit d'intervertir les étiquettes L_{true} et L_{false}.

Définition orientée-syntaxe:

Production	Règles sémantiques
$B \rightarrow$ **not** B_1	$B_1.L_{true} := B.L_{false}$ $B_1.L_{false} := B.L_{true}$ $B.ccode := B_1.ccode$

Traduction d'expressions avec court-circuit: constantes booléennes

Exemple de code source:

```
if (true) then
    ...
else
    ...
```

Allure du code généré:

```
    goto Ltrue // branche toujours à Ltrue

    Ltrue:
        ...
    Lfalse:
        ...
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$B \rightarrow$ **true**	$B.ccode := gen(\text{`}\textbf{goto}\text{'}\ B.L_{true})$
$B \rightarrow$ **false**	$B.ccode := gen(\text{`}\textbf{goto}\text{'}\ B.L_{false})$

Traduction d'expressions avec court-circuit: accès à une variable

Exemple de code source:

```
if (x) then
    ...
else
    ...
```

Allure du code généré:

```
if x.place ≠ 0 goto Ltrue
goto Lfalse

Ltrue:
    ...
Lfalse:
    ...
```

Définition orientée-syntaxe:

PRODUCTION	RÈGLES SÉMANTIQUES
$B \rightarrow$ **id**	$B.ccode := gen(\text{`}\textbf{if}\text{'}\ \textbf{id}.place\ \text{`}\neq\ 0\ \textbf{goto}\text{'}\ B.L_{true}) \parallel gen(\text{`}\textbf{goto}\text{'}\ B.L_{false})$

Traduction d'expressions et d'énoncés avec court-circuit

Observation: Le code à court-circuit que nous avons généré aux pages 22–24 et 27–31 **n**'est **pas** d'aussi bonne qualité que celui suggéré à la page 13.

Le code que nous générons pour une expression booléenne requiert un saut vers toute destination (e.g. la branche `then`) même lorsque cette destination se trouve immédiatement après l'expression.

Il y a moyen d'améliorer la génération du code pour les expressions booléennes dans le cas de sauts vers du code qui suit immédiatement une expression booléenne.

La section 6.6.5 du manuel aborde cette question.

Il s'agit de se munir d'une étiquette spéciale "*fall*" qui signifie qu'on doit laisser le contrôle filer tout droit lorsque la destination se trouve immédiatement après.

Traduction d'expressions et d'énoncés

Principe: En génération de code, il faut éviter la *duplication* du code.

En d'autres mots, soient C, C_1, ..., C_k des non-terminaux générant diverses constructions syntaxiques (par exemple, des expressions, des énoncés) et soit la production $C \rightarrow \alpha_0 C_1 \alpha_1 \ldots C_k \alpha_k$, alors $C.code$ doit être formé en intégrant au plus une copie de chaque $C_i.code$, pour $1 \leq i \leq k$.

En présence de duplication de code, on court le risque d'une explosion de la taille du code généré.

Exemple: Supposons que nous avons une expression dotée d'un opérateur binaire $\oplus$, générée par $E \rightarrow E_1 \oplus E_2$ et telle que

$$E.code \quad := \quad \ldots \parallel E_1.code \parallel \ldots \parallel E_2.code \parallel \ldots \parallel E_2.code \parallel \ldots$$

C'est-à-dire que *deux* copies du code de E_2 sont intégrées dans le code de E. Alors,

le code de $E_1 \oplus E_2$	contient	2 copies du code de E_2,
le code de $E_1 \oplus (E_2 \oplus E_3)$	contient	4 copies du code de E_3,
le code de $E_1 \oplus (E_2 \oplus (E_3 \oplus E_4))$	contient	8 copies du code de E_4,
...		
le code de $E_1 \oplus (E_2 \oplus \ldots (E_{k-1} \oplus E_k) \ldots)$	contient	2^{k-1} copies du code de E_k.

Analyse Sémantique:

Vérification de la sémantique

Types et déclarations

Sections 6.3 et 6.5

Outre le fait de vérifier la syntaxe des programmes, le compilateur doit aussi vérifier le respect de certaines conventions liées à leur sémantique.

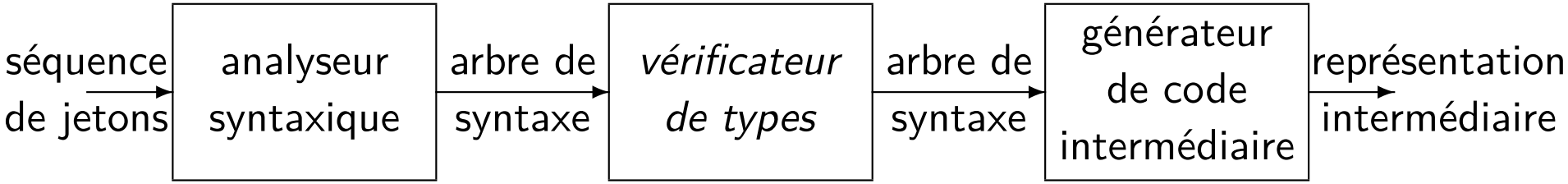
Ces vérifications s'appellent les *vérifications statiques*, par opposition aux *vérifications dynamiques*, qui se font à l'exécution des programmes.

Voici certaines sortes de vérifications statiques:

- *Vérification des types*.
- *Vérifications liées au flôt de contrôle*. Par exemple: y a-t-il un '`while`' ou un '`switch`' entourant le '`break`' courant?
- *Vérifications d'unicité*. Par exemple: est-ce que la même variable est déclarée deux fois?
- *Vérifications liées à l'utilisation des noms*. Par exemple: en Ada, un bloc doit être nommé à la fois à son ouverture et à sa fermeture; les noms correspondent-ils?

Types et déclarations

Cette section se concentre sur la vérification des types.



Des opérations habituelles de vérification de types incluent, par exemple, le fait de vérifier: que les opérandes de '`mod`' sont entiers; qu'un accès indirect à la mémoire se fait à l'aide d'un pointeur; qu'un accès indexé se fait dans un tableau; qu'une fonction est appliquée au bon nombre d'arguments et que ceux-ci sont du bon type.

Parmi les opérations plus spéciales, il y a le typage d'opérateurs *surchargés*. Par exemple: '+'. Des conversions automatiques peuvent être ajoutées lors du typage.

Aussi, il y a la notion de *polymorphisme*, c'est-à-dire des fonctions qui peuvent recevoir et manipuler des valeurs de différents types indifféremment. Par exemple: la fonction identité.

Systèmes de types concrets

Le design d'un vérificateur doit habituellement être fait à partir des spécifications informelles du langage de programmation.

Exemples d'extraits de spécifications:

- "Si les deux opérandes des opérateurs arithmétiques d'addition, de soustraction et de multiplication sont de type entier, alors le résultat est de type entier."
- "Le résultat de l'opérateur unaire '`&`' est un pointeur vers l'objet désigné par l'opérande. Si le type de l'opérande est τ, alors le type du résultat est 'pointeur de τ'."

Ces extraits sous-entendent que chaque expression a un type.

Dans la plupart des langages, on retrouve des *types de base* (par exemple: booléens, caractères, entiers, réels) et des *types construits* (par exemple: tableaux, structures, ensembles, pointeurs, fonctions).

Expressions de types

Nous définissons les *expressions de types*, lesquelles servent à dénoter des types.

Une expression de types est soit un type de base, soit un type construit grâce à l'application d'un *constructeur de types* à d'autres types.

1. Un *type de base* comme "entier" est une expression de type. On ajoute les types spéciaux "*void*" pour les énoncés et "*type_error*" lorsqu'on détecte une erreur de type.

2. Le nom d'un *type nommé* est une expression de types. Par exemple, en Java, le nom d'une classe est une expression de types.

3. (suite à la page suivante)

Expressions de types

3. Un *constructeur de types* appliqué à des expressions de types constitue une expression de types. Si on considère que τ, τ_1, ..., τ_n sont des expressions de types, les constructions suivantes sont des expressions de types:

   (a) Les *tableaux*. $array(I, \tau)$ est le type d'un tableau d'éléments de type τ et où les index permettant d'accéder au tableau sont dans I (un intervalle).

   (b) Les *produits cartésiens*. $\tau_1 \times \tau_2$ est une expression de types. $\times$ est associatif à gauche.

   (c) Les *structures* (ou *enregistrements*). Une structure est comme où tuple mais où les champs portent des noms. $record((\mathbf{id}_1 \times \tau_1) \times \ldots \times (\mathbf{id}_n \times \tau_n))$ est une expression de types.

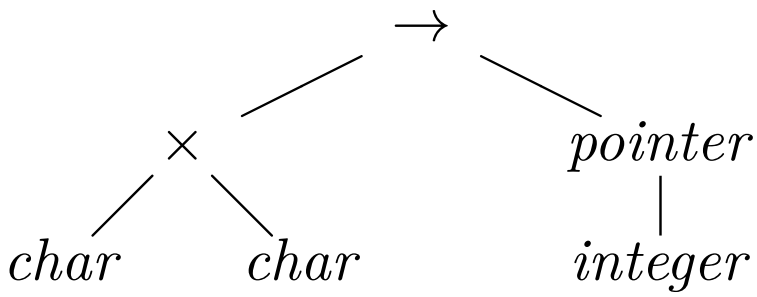
   (d) Les *pointeurs*. $pointer(\tau)$ est une expression de types.

   (e) Les *fonctions*. $\tau_1 \rightarrow \tau_2$ est une expression de types pour les fonctions où τ_1 est appelé le *domaine* et τ_2 est appelé l'*image*.

4. Dans certains systèmes de typage, on permet l'existence de *variables de types*. Celles-ci sont utilisées lorsqu'on a des *définitions* de types *récursives* ou lorsqu'on a du *polymorphisme*.

Vérification des types

À l'instar des programmes analysés syntaxiquement qui ont un arbre de syntaxe, les types peuvent être représentés sous forme d'arbres. Par exemple, voici l'arbre correspondant au type $char \times char \rightarrow pointer(integer)$:



Un *système de types* est un ensemble de règles qui permettent d'assigner des types aux diverses composantes d'un programme.

Vérification des types

Un *vérificateur de types* est une implantation d'un système de types.

Une vérification des types faite par le compilateur est appelée *statique* tandis qu'une vérification faite à l'exécution par le programme lui-même est appelée *dynamique*. La vérification dynamique ajoute une pénalité en temps à l'exécution.

Un système de types statique est *bien fondé* s'il élimine le besoin de vérifier les types dynamiquement en effectuant toutes les vérifications à la compilation.

Un langage est *fortement typé* si le compilateur est en mesure de garantir (pour un programme jugé correct) qu'il ne causera jamais d'erreurs de types à l'exécution.

Un compilateur complet doit détecter et signaler les erreurs de types. Autant que possible, les messages doivent être clairs et informatifs.

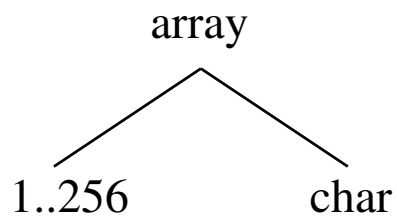
Un vérificateur de types

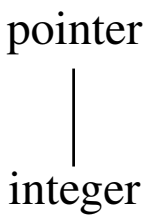
Considérons la syntaxe suivante pour nos programmes:

$$
\begin{array}{lcl}
P & \rightarrow & D \ ; \ E \\
D & \rightarrow & D \ ; \ D \mid \mathbf{id} : T \\
T & \rightarrow & \mathbf{char} \mid \mathbf{integer} \mid \mathbf{array} \ [\ \mathbf{num} \ ] \ \mathbf{of} \ T \mid \uparrow T \mid \ldots \\
E & \rightarrow & \mathbf{literal} \mid \mathbf{num} \mid \mathbf{id} \mid E \ \mathbf{mod} \ E \mid E \ [\ E \ ] \mid E \uparrow \mid E \ == \ E \mid \ldots
\end{array}
$$

Les expressions de types que nous utiliserons pour faire la vérification des types des programmes incluent les types de base *char*, *integer* et *type_error* ainsi que les tableaux et les pointeurs.

Exemples de types: "`array [256] of char`" et "↑ `integer`".





Un vérificateur de types

Typage des déclarations:

P	$\rightarrow$	$D \; ; \; E$	
D	$\rightarrow$	$D \; ; \; D$	
D	$\rightarrow$	$\mathbf{id} \; : T$	$\{addtype(\mathbf{id}.entry, T.type)\}$
T	$\rightarrow$	$\mathbf{char}$	$\{T.type := char\}$
T	$\rightarrow$	$\mathbf{integer}$	$\{T.type := integer\}$
T	$\rightarrow$	$\uparrow T_1$	$\{T.type := pointer(T_1.type)\}$
T	$\rightarrow$	$\mathbf{array} \; [\; \mathbf{num} \;] \; \mathbf{of} \; T_1$	$\{T.type := array(1..\mathbf{num}.val, T_1.type)\}$
T	$\rightarrow$	$\ldots$	

Un vérificateur de types

Typage des expressions:

$$
\begin{array}{lcll}
E & \rightarrow & \textbf{literal} & \{E.type := char\} \\
E & \rightarrow & \textbf{num} & \{E.type := integer\} \\
E & \rightarrow & \textbf{id} & \{E.type := lookup(\textbf{id}.entry)\} \\
E & \rightarrow & E_1 \ \textbf{mod} \ E_2 & \{E.type := \textbf{if} \ E_1.type = integer \ \textbf{and} \\
 & & & \quad E_2.type = integer \ \textbf{then} \ integer \\
 & & & \textbf{else} \ type\_error\} \\
E & \rightarrow & E_1 \ [\ E_2 \ ] & \{E.type := \textbf{if} \ E_1.type = array(s,t) \ \textbf{and} \\
 & & & \quad E_2.type = integer \ \textbf{then} \ t \\
 & & & \textbf{else} \ type\_error\} \\
E & \rightarrow & E_1 \uparrow & \{E.type := \textbf{if} \ E_1.type = pointer(t) \ \textbf{then} \ t \\
 & & & \textbf{else} \ type\_error\} \\
E & \rightarrow & E_1 == E_2 & \{E.type := \textbf{if} \ E_1.type = E_2.type \neq type\_error \\
 & & & \textbf{then} \ boolean \\
 & & & \textbf{else} \ type\_error\} \\
E & \rightarrow & \ldots &
\end{array}
$$

Un vérificateur de types

Typage des énoncés:

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} := E & \{S.type := \textbf{if } \textbf{id}.type = E.type \textbf{ then } void \\
 & & & \quad \textbf{else } type\_error\} \\
S & \rightarrow & \textbf{if } E \textbf{ then } S_1 & \{S.type := \textbf{if } E.type = boolean \textbf{ then } S_1.type \\
 & & & \quad \textbf{else } type\_error\} \\
S & \rightarrow & \textbf{while } E \textbf{ do } S_1 & \{S.type := \textbf{if } E.type = boolean \textbf{ then } S_1.type \\
 & & & \quad \textbf{else } type\_error\} \\
S & \rightarrow & S_1 \ ; \ S_2 & \{S.type := \textbf{if } S_1.type = void \textbf{ and} \\
 & & & \quad S_2.type = void \textbf{ then } void \\
 & & & \quad \textbf{else } type\_error\}
\end{array}
$$

où un énoncé correct a le type $void$, sinon le type $type\_error$.

Un vérificateur de types

Si on ajoute les fonctions et les appels de fonctions, on doit pouvoir construire les types reliés aux fonctions grâce à la production suivante:

$$T \quad \rightarrow \quad T_1 \text{ '}\rightarrow\text{' } T_2 \quad \{T.type := T_1.type \rightarrow T_2.type\}$$

et faire le typage des appels de fonctions:

$$\begin{aligned} E \quad \rightarrow \quad E_1 \ (\ E_2\ ) \quad \{E.type := & \textbf{ if } E_1.type = s \rightarrow t \textbf{ and} \\ & E_2.type = s \textbf{ then } t \\ & \textbf{else } type\_error\} \end{aligned}$$

Déclarations et disposition des données

Le compilateur doit déterminer la disposition des données globales et locales. (Plus généralement, dans le cas des déclarations locales, le compilateur doit déterminer la disposition de l'ensemble du contenu des structures d'activation; voir Chapitre 7).

L'espace qu'on doit prévoir pour une variable donnée dépend de son type.

Chacune des variables se voit assigner une *position relative* à une adresse symbolique représentant le début de la région qui accommode les variables.

Sur la plupart des machines, il y a des contraintes d'*alignement* sur les données. Par exemple, un nombre en point flottant (`double`) peut requérir un alignement sur 64 bits.

Si la prochaine position relative n'est pas alignée correctement pour effectuer l'allocation de la prochaine variable, on ajoute des octets de *remplissage*, lesquels ne servent pas à contenir de l'information utile.

Lorsque le langage le permet, un compilateur optimisant peut effectuer un réordonnancement des variables de telle façon qu'on requiert moins d'octets de remplissage.

Déclarations et disposition des données

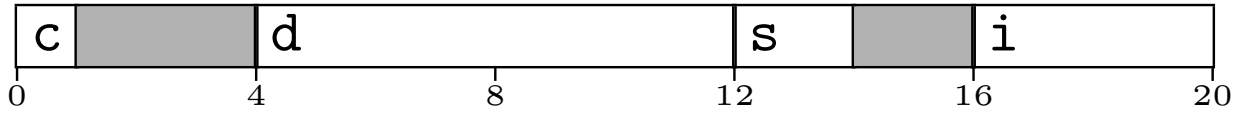
Exemple

Déclarations:

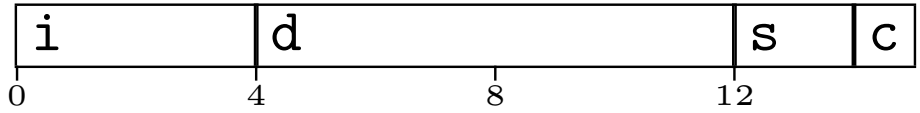
```
char c;
double d;
short s;
int i;
```

Type	Taille	Alignement
char	1	1
short	2	2
int	4	4
double	8	4

Disposition sans réordonnancement:



Disposition avec réordonnancement permis:



Déclarations et disposition des données

Exemple. On cherche à accumuler les détails du stockage des variables à partir de leurs déclarations.

- La méthode est présentée à la page suivante.
- La méthode présentée est appropriée pour des variables telles que celles utilisées dans des langages comme C, Pascal et FORTRAN.
- On reprend les hypothèses de la page précédente sur la taille et l'alignement des types.
- L'opérateur d'alignement est défini ainsi: $o \triangleright a = a \times \left\lceil \frac{o}{a} \right\rceil$.
- On doit employer des techniques similaires (mais plus lourdes) pour traiter le cas des procédures imbriquées et des enregistrements. On omet leur présentation, ici.

Déclarations et disposition des données

Exemple. (suite)

Productions	Règles sémantiques
$P \to D$	$D.offsetI := 0$
$D \to D_1 \ ; \ D_2$	$D_1.offsetI := D.offsetI$ $D_2.offsetI := D_1.offsetS$ $D.offsetS := D_2.offsetS$
$D \to \textbf{id} : T$	$enter(\textbf{id}.name, T.type, D.offsetI \triangleright T.align)$ $D.offsetS := D.offsetI \triangleright T.align + T.width$
$T \to \textbf{char}$	$T.type := char$ $T.width := 1; \quad T.align := 1$
$T \to \textbf{short}$	$T.type := short$ $T.width := 2; \quad T.align := 2$
$T \to \textbf{integer}$	$T.type := integer$ $T.width := 4; \quad T.align := 4$
$T \to \textbf{long}$	$T.type := long$ $T.width := 8; \quad T.align := 4$
$T \to \textbf{float}$	$T.type := float$ $T.width := 4; \quad T.align := 4$
$T \to \textbf{double}$	$T.type := double$ $T.width := 8; \quad T.align := 4$
$T \to \textbf{array [num] of } T_1$	$T.type := array(1..\textbf{num}.val, T_1.type)$ $T.width := \textbf{num}.val \times T_1.width$ $T.align := T_1.align$
$T \to \uparrow T_1$	$T.type := pointer(T_1.type)$ $T.width := 4; \quad T.align := 4$