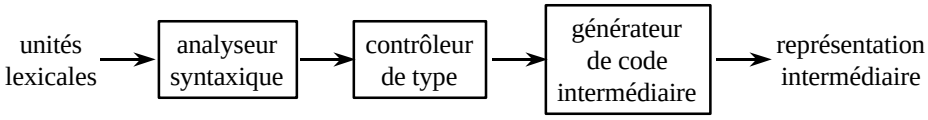
CH.5 LE CONTRÔLE DE TYPE

- 5.1 Les systèmes de typage
- 5.2 Un contrôleur de type simple
- 5.3 L'équivalence des types
- 5.4 La manipulation des types

5.1 Les systèmes de typage

Font partie de l'analyse sémantique, plus précisément du **contrôle statique :**

- **Le contrôle de type :**
  incompatibilité de type entre opérandes ;
- **Le contrôle du flot d'exécution :**
  transfert du contrôle en un autre point du programme ;
- **Le contrôle d'unicité ou de répétition :**
  un nom peut devoir apparaître un nombre fixé de fois.



En une passe (PASCAL) ou plusieurs (ADA), si surcharges ou polymorphisme élaborés.

Le contrôle et la manipulation des types font partie des spécifications du langage (voir manuels de référence).

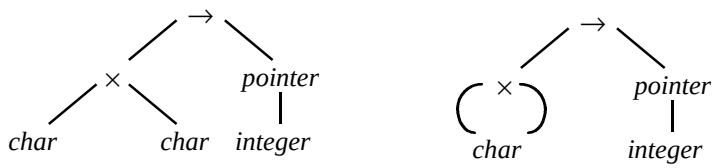
Expressions de type :

- Type de base : *boolean, char, integer, real, ...erreur, void*
- Noms de type (par exemple pour la récursion)
- Constructeurs de type :
  - (a) Les tableaux : *array*(*I*,*T*) ;
  - (b) Les produits : $T_1 \times T_2$, paires d'objets;
  - (c) Les enregistrements : produits avec noms de champs ;
  - (d) Les pointeurs : *pointer*(*T*) ;
  - (e) Les fonctions : $D \rightarrow R$ fonctions de D vers R.
- Variables pour définir des expressions de type.

Exemple :

Une expression de type représentée par un arbre abstrait ou par un graphe orienté acyclique.

$$char \times char \rightarrow pointer(integer)$$



Représente le type d'une fonction à deux arguments de type *char* retournant une valeur de type pointeur vers entier.

Systèmes de typage :

Mis en œuvre par le contrôleur de type. Peuvent dépendre de l'implémentation. L'ensemble des indices d'un tableau fait-il partie du type de ce tableau ?
Certains outils logiciels (*lint*) utilisent un système de typage plus strict pour des motifs de débogage.

Contrôles statique et dynamique :

Le premier effectué par le compilateur, le second à l'exécution.
Certains contrôles ne peuvent être faits que dynamiquement :

```
table : array[0..255] of char ; i : integer ; ... table[i] ...
```

En général ne peut être contrôlé statiquement.
Langage fortement typé : aucune erreur de type à l'exécution.

Récupération sur erreur :

Même technique de récupération que pour l'analyse syntaxique.

5.2 Un contrôleur de type simple

Syntaxe d'un langage rudimentaire :

$P \rightarrow D$; **begin** I **end**
$D \rightarrow D$; D | **id** : T
$T \rightarrow$ **char** | **int** | **boolean** | **array[nb] of** T | $\uparrow T$
| **function** T **to** T
$E \rightarrow$ **literal** | **nb** | E **=** E | E**(**E**)** | F
$F \rightarrow$ **id** | E**[**E**]** | $E\uparrow$
$I \rightarrow F$ **:=** E | **if** E **then** I **fi** | **while** E **do** I **od** | I ; I

Exemple de programme :

```
clef : ↑int ; ok : boolean;
begin ok := clef ↑ = 2000 ; if ok then clef↑ := 1 end
```

La grammaire est ambiguë. On peut utiliser des règles d'associativité. Le bloc **begin...end** permet de lever l'ambiguïté entre déclarations et instructions. La variable *F* désigne les expressions assignables

Type des identificateurs :
au moyen d'un attribut synthétisé *T.type.*

$P \rightarrow D$; **begin** I **end**	
$D \rightarrow D$; D	
$D \rightarrow$ **id** : T	{*addtype*(**id**.*ent, T.type*)}
$T \rightarrow$ **char**	{*T.type* := *char*}
$T \rightarrow$ **int**	{*T.type* := *int*}
$T \rightarrow$ **boolean**	{*T.type* := *boolean*}
$T \rightarrow$ **array[nb] of** T	{*T.type* := *array*(*num.val*,T_1.*type*)}
$T \rightarrow \uparrow T$	{*T.type* := *pointer*(T_1.*type*)}
$T \rightarrow$ **function** T **to** T	{*T.type* := T_1.*type* $\rightarrow$ T_2.*type*}

La procédure *addtype*(**id**.*ent, T.type*) ajoute l'information relative au type de l'identificateur dans la table des symboles.

Type des expressions : au moyen d'un attribut synthétisé .

$E \rightarrow$ **literal**	{*E.type* := *char*}
$E \rightarrow$ **nb**	{*E.type* := *int*}
$E \rightarrow E = E$	{*E.type* := **if** E_1.*type* = E_2.*type* <> *erreur* **then** *boolean* **else** *erreur*}
$E \rightarrow E$**(**E**)**	{*E.type* := **if** E_2.*type* = *s* **and** E_1.*type* = *s* $\rightarrow$ *t* **then** *t* **else** *erreur*}
$E \rightarrow F$	{*E.type* := *F.type*}
$F \rightarrow$ **id**	{*F.type* := *lookup*(**id**.*ent*)}
$F \rightarrow E$**[**E**]**	{*F.type* := **if** E_2.*type* = *int* **and** E_1.*type* = *array*(*s*, *t*) **then** *t* **else** *erreur*}
$F \rightarrow E\uparrow$	{*F.type* := **if** E_1.*type* = *pointer*(*t*) **then** *t* **else** *erreur*}

La fonction *lookup*(**id**.*ent*) retourne le type de l'identificateur trouvé dans la table des symboles. On ne vérifie pas la validité de l'indice dans le tableau (contrôle dynamique).

Type des instructions :
au moyen d'un attribut synthétisé *I.type.*

$I \rightarrow F$ **:=** E	{*I.type* := **if** *F.type* = *E.type* **then** *void* **else** *erreur*}
$I \rightarrow$ **if** E **then** I **fi**	{*I.type* := **if** *E.type* = *boolean* **then** I_1*.type* **else** *erreur*}
$I \rightarrow$ **while** E **do** I **od**	{*I.type* := **if** *E.type* = *boolean* **then** I_1*.type* **else** *erreur*}
$I \rightarrow I$; I	{*I.type* := **if** I_1*.type* = I_2*.type* = *void* **then** *void* **else** *erreur*}

Les fonctions à plusieurs arguments peuvent être traitées au moyen des types produits.

5.3 L'équivalence des types

Quand deux expressions de type sont-elles égales ?
Que représente un nom de type ?
Interaction avec la représentation des types :
Arbres abstraits dont les feuilles sont des types de base ou graphes (usage limité en C).

Équivalence structurelle :
égalité des expressions, définie récursivement. Ne permet pas de rendre compte des cycles. Peut être affaiblie (en ne tenant pas compte des dimensions des tableaux passés en paramètres, par exemple.)

Équivalence nominale :
identité des noms de type.

Équivalence structurelle :

function *equiv*(*s*,*t*) : **boolean** ;
begin
if *s* et *t* sont du même type de base **then** *equiv* := **true**
else if $s = s_1 \times s_2$ **and** $t = t_1 \times t_2$ **then**
equiv := (*equiv*(s_1 ,t_1) **and** *equiv*(s_2 ,t_2))
else if $s = array(s_1,s_2)$ **and** $t = array(t_1,t_2)$ **then**
equiv := *equiv*(s_2 ,t_2)
et ainsi de suite.

On ignore ici l'ensemble des indices pour l'équivalence structurelle des tableaux.

Équivalence nominale :

On peut souvent nommer les types :

```
type lien = ↑ cellule ;
var suivant : lien ;
    dernier : lien ;
    p : ↑ cellule ;
    q, r : ↑ cellule ;
```

Les variables suivant, dernier, p, q et r ont-elles le même type ?
Cela dépend du type d'équivalence.
Pas de réponse unique dans la norme PASCAL.

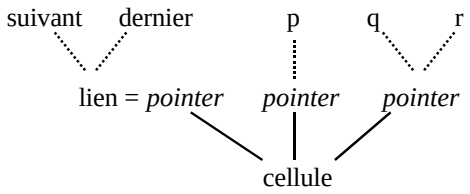
Diverses possibilités :

Structurellement, toutes sont équivalentes.
En équivalence par nom, suivant et dernier sont équivalentes, ainsi que p, q et r ; par contre, p et suivant ne le sont pas.
Mais de nombreuses implémentations de PASCAL créent des noms de types implicites lors des déclarations, et tout se passe comme si on avait :

```
type lien = ↑cellule ;
       np = ↑cellule ;
       nq = ↑cellule ;
var suivant : lien ;
      dernier : lien ;
      p : np ;
      q : nq ;
      r : nq ;
```

Dans ce cas, p et q ne sont plus du même type.

L'implémentation classique est faite au moyen d'un graphe de type. On crée un nœud pour chaque constructeur et chaque type de base, et une feuille pour chaque nom. Dans l'exemple, cela donne :

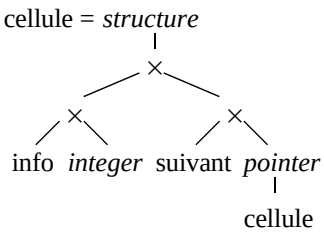


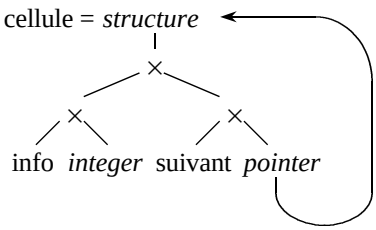
Les cycles dans les représentations de types :

Définitions récursives possibles en PASCAL.

```
type lien = ↑cellule ;
      cellule = record
                    info : integer ;
                    suivant : lien
      end ;
```

Deux représentations possibles :
la première, sans cycle en dupliquant un nom ;
la seconde, avec cycle .





En C est utilisée l'équivalence structurelle sauf pour les types structurés. Tous les noms de types doivent être déclarés, sauf les pointerrus vers des structures non encore déclarées. Les seules récursivités proviennent donc des structures. On peut donc utiliser la représentation acyclique. L'équivalence structurelle s'arrête lorsqu'on atteint un constructeur de structure ; dans ce cas, ou bien ils portent le même nom et sont déclarés équivalents, ou bien ils ne le sont pas.

```
struct cellule {
   int info ;
   struct cellule *suivant ;
} ;
```

5.4 La manipulation des types

Conversion de type nécessaire dans de nombreuses situations, par exemple **x + i** lorsque le premier opérande est un réel et le second un entier.

Ces conversions peuvent être :

• explicites = application d'une fonction, comme ord en PASCAL ;

• implicites = **coercition**, par exemple entier vers réel.

Dans ce dernier cas, il y a souvent **surcharge** des opérateurs.

Dans l'exemple, cela peut être pris en charge simplement au niveau du schéma de traduction :

$E \rightarrow E+E$ [*E.val, E.type*] := **if**
$\boldsymbol{E_1}$*.type = real* **and** $\boldsymbol{E_2}$*.type = integer* **then**
[*realadd*($\boldsymbol{E_1}$*.val*,*inttoreal*($\boldsymbol{E_2}$*.val*)), *real*]
.....

La conversion des constantes peut se faire à la compilation = gain de temps à l'exécution.

Dans l'exemple précédent, le symbole + était surchargé.

On peut parfois **résoudre** la surcharge en examinant les opérandes (cas précédent). C'est parfois impossible.

Exemple : (inspiré d'ADA)

L'opérateur * est une fonction d'un couple d'entiers qui retourne une valeur entière. On peut le surcharger :

```
function "*" (i, j : integer) return complex ;
function "*" (i, j : complex) return complex ;
```

Les types possibles pour * sont donc :

$integer \times integer \rightarrow integer$	$i \times i \rightarrow i$
$integer \times integer \rightarrow complex$	$i \times i \rightarrow c$
$complex \times complex \rightarrow complex$	$c \times c \rightarrow c$

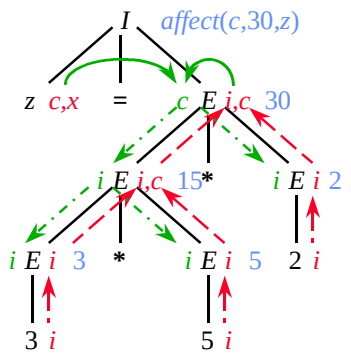
Si 2, 3 et 5 sont des entiers, le type de 3*5 dépend du contexte :

integer dans 2*(3*5), *complex* dans (3*5)*z*

On suppose que le type est fixé lors d'une affectation.
La grammaire attribuée suivante réalise la réduction en plusieurs passages : les types possibles, **.types*, attributs synthétisés évalués pendant l'analyse ascendante, les types uniques, **.u*, attributs hérités en descendant l'arbre, et l'action finale, **.val* ou *affect*, attributs synthétisés en remontant.
On pose *mult.types* := $\{i \times i \to i,\ i \times i \to c,\ c \times c \to c\}$et on suppose que *mult*(*s*, *x*, *y*) effectue la multiplication relative à *s* dans *mult.types* avec opérandes *x* et *y*.

$I \to$ **id** = E	$E.u$:= **if** (**id**.*types* $\cap$ $E.types$) = t **then** t **else** *erreur*
	affect($E.u$, $E.val$, **id**.*adr*)
$E \to$ **nb**	$E.types$:= **nb**.*types* ; $E.val$:= *convert*($E.u$, **nb**.*val*)
$E \to E*E$	$E.types$:= $\{t : \exists\ s, (s \in E_1.types \times E_2.types) \wedge (s \to t \in mult.types)\}$
	u := $E.u$; S := $\{s : (s \in E_1.types \times E_2.types) \wedge (s \to u \in mult.types)\}$
	$[E_1.u, E_2.u]$:= **if** $S = \{[s_1, s_2]\}$ **then** $[s_1, s_2]$ **else** *erreur*
	$E.val$:= *mult*($[E_1.u, E_2.u] \to E.u, E_1.val, E_2.val$)

Traduction de $z = 3*5*2$,
z est soit complexe, soit autre x, non entier.



En montant, les ensembles de types possibles.
En descendant, le type unique hérité.
A droite, la valeur calculée, du type unique associé.

ADA exige que chaque expression ait un type unique déterminé avant l'évaluation.