INSA ROUEN

Génie Mathématique – 3ème année

Cours Langage C

(Annexes)

KOTOWICZ JEAN-PHILIPPE

(jean-philippe.kotowicz@insa-rouen.fr)

VERSION 1.1

Table des matières

Annexe 1 : Le pré-processeur

1. Principes

Une particularité propre au compilateur C est qu'il ne procède pas directement sur le code source fournit par le programmeur. Une phase spéciale de réécriture du programme précède toute compilation. L'utilitaire se chargeant de cette phase de pré-traitement se nomme le pré-processeur.

Attention : le programme n'est pas compilé durant le traitement du pré-processeur. Si des erreurs existent, elles ne seront pas déterminées dès lors. Il s'agit juste de réécrire le programme selon des règles que vous précisez.

1.1. Syntaxe des commentaires

Les commentaires sont délimités par **/*** (pour marquer le début du commentaire) et ***/** (pour en marquer la fin).
Ils peuvent s'étendre sur plusieurs lignes mais ne peuvent en aucun cas être imbriqués (mettre un commentaire dans un commentaire).
Ils doivent en outre apparaître entre les entités lexicales.

Exemple :

```
/* exemple de commentaire rédigé
sur plusieurs lignes
*/
n= Nmax; /* commentaire bref */
```

Tous les commentaires seront supprimés lors de la phase de pré-processing.

1.2. Syntaxe des directives

Les directives de pré-compilation commencent en colonne 1.
Elles utilisent une ligne complète et peuvent s'étendre sur plusieurs lignes. Dans ce cas, il faut indiquer au moyen d'un antislash (le caractère '\') que la directive continue sur la ligne suivante.

La syntaxe est donc la suivante :

#mot_clé_en_minuscule

Nous allons voir dans la suite de ce chapitre les mots clés reconnus par le pré-processeur, en attendant, en voici quelques exemples :

```
#include "monFichier.h"
#define LgMax \
80
```

Attention à ne **jamais** mettre de '**;**' à la fin d'une directive de pré-compilation.

Dans le fichier issu de l'exécution du pré-processeur, les directives auront disparues, elles auront été remplacées par le résultat de l'exécution de ces directives.
Voyons donc quelles sont ces directives et ce qu'elles permettent.

2. Les directives du pré-processeur

2.1. Inclusion de fichiers

L'inclusion de fichiers par le pré-processeur est déclenchée par la rencontre de la directive :

#include "*nom_de_fichier****"***

où *nom_de_fichier* est le nom sous lequel le fichier a été sauvegardé (donc avec son extension s'il en a une).

Cette instruction permet donc d'inclure un fichier dans celui en cours de traitement. Cette instruction prend en unique paramètre le nom du fichier à inclure. Cette inclusion est ponctuelle : la totalité du fichier à inclure se retrouve donc entre la ligne précédent l'instruction d'inclusion et la suivante. Le compilateur, lui, n'aura plus qu'à traiter un gros fichier résultant de la fusion des deux fichiers initiaux.
Les fichiers inclus sont traités par le pré-processeur. Ils peuvent contenir d'autres inclusions de fichiers. Ce processus récursif est parfois limité à quelques niveaux d'inclusion.

Il vous est possible d'entourer le nom du fichier par les signes **< >,** plutôt que par des guillemets. Le fichier sera alors recherché uniquement dans les répertoires contenants les fichiers standard fournis avec le compilateur.
Lorsque le nom du fichier est entre guillemets, le fichier est cherché d'abord dans le répertoire courant (celui dans lequel vous vous trouvez), et si le fichier n'existe pas, alors le pré-processeur cherchera dans les répertoires standard.

Exemple :

```
#include "fichier.h"    /* pour un fichier du répertoire courant */
#include <fichier.h>    /* pour un fichier standard */
#include "/usr/local/machin/bidule.h"
#include <stdio.h>
```

Parmi les bibliothèques standards, on utilise couramment : *stdio.h math.h string.h stdlib.h*, …

2.2. Substitution de texte

Le pré-processeur permet de réaliser des substitutions de texte et ainsi de définir des pseudo-constantes et de faire de la macro-génération de code.

2.2.1 Substitution sans paramètre

La substitution de code sans paramètre permet de définir des **constantes de compilation**. De telle directive se rédigent comme suit :

#define *identificateur texte_de_remplacement*

L'usage le plus courant des constantes de compilation est associé à la manipulation de tableaux. Il est plus simple et plus sûr d'avoir une constante qui soit utilisée lors de la définition et lors de la manipulation du tableau.
Ceci évite de rechercher dans le fichier source les instructions qui font référence à la taille du tableau lorsque cette taille change.
Lorsqu'une constante de compilation a été définie, le pré-processeur remplace toutes les occurrences du nom de la constante par sa valeur, sauf lorsque le nom se trouve dans une chaîne de caractères. Le changement ne se fait que lorsque le nom de la variable est isolé.

Exemple :

```
#define LgMax  80
#define LgMaxP2  (LgMax+2)
#define  fois  *

If (lg < LgMax )
   nb = 3 fois LgMaxP2;
```

après passage du pré-processeur, le fichier obtenu sera le suivant :

```
If ( lg < 80 )
    nb = 3 * ( 80 + 2 )
```

2.2.2 Substitution avec paramètres

De la même manière, on peut, pour une substitution de code, préciser des paramètres en utilisant l'instruction #define suivie du nom de la macro-fonction (ou macro-expression), d'une parenthèse ouvrante, de la liste des arguments, d'une parenthèse fermante, et de la définition du corps de la macro :

#define *identificateur (id1, id2, …) texte*

Exemple :

```
#define carre(x)   ((x)*(x))
#define double(x)  ((x)+(x))
#define Theta 5

… y = carre(double(Theta)); ….
```

après passage du pré-processeur, le fichier obtenu sera le suivant :

```
Y = ((((5)+(5)))*(((5)+(5))));
```

Il faut faire attention à l'ordre d'évaluation pour des macro-expressions complexes, et la technique la plus simple est d'encadrer le nom des pseudo-variables dans l'expression de la macro. Dans tous les cas, le parenthésage garantit la bonne interprétation de l'expression.

Par exemple, si on avait définie la macro carre par #define carre(x) (x*x)

De plus, la définition de macros est récursive et les directives de substitution du pré-compilateur seront appliquées tant qu'il reste des macros définies à remplacer.

```
#define max(a,b)  (a>=b?a:b)
#define min(a,b)  (a<=b?a:b)

max(min(a,b),c)    produira      ((a<=b?a:b)>=c?(a<=b?a:b):c)
                   et non pas    (min(a,b)>=c?min(a,b):c)
```

2.2.3 Portée des substitutions

On peut également limiter la portée des substitutions par la directive :

#undef *identificateur*

La variable est alors considérée comme non définie. Dans le cas où la variable est associée à une valeur, les occurrences de son nom ne sont plus remplacées par sa valeur.

Exemple :

```
Y = Rho2 + R(5);
#define Rho2  R(2)
Y = Rho2 + R(5);
#define R(x)  ((x)-1)
Y = Rho2 + R(5);
#undef R
Y = Rho2 + R(5);
```

après passage du pré-processeur, le fichier obtenu sera le suivant :

```
Y = Rho2 + R(5);
Y = R(2) + R(5);
Y = ((2)-1)+((5)-1);
Y = Rho2 + ((5) -1);
Y = R(2) + R(5);
```

La norme ANSI dit que si il y a redéfinition de la macro, alors il y a remplacement de la définition. Certains pré-processeurs signalent tout de même la redéfinition par un petit message. Certains autres pré-processeurs admettent même une pile de définition, permettant ainsi de restituer les anciennes définitions lors de l'utilisation de l'instruction ***#undef*** (mais l'existence d'une telle pile de définitions ne fait pas partie de la norme ANSI).

2.2.4 Les macros prédéfinies

Un certain nombre de macros sont prédéfinies. Ces macros ne peuvent en aucun cas être supprimées ou altérées. Elles sont donc immuables. Le tableau suivant vous donne le nom de quelques macros ainsi que leur développement.

Nom des macros	Développement de la macro associée.
__LINE__	Se développe en une valeur numérique associée au numéro de la ligne courante dans le code du programme
__FILE__	Cette macro sera remplacée par le nom du fichier en cours de traitement
__DATE__	Celle-ci se transformera en une chaîne de caractères contenant la date de traitement du fichier sous un format "mm jj aaaa"
__TIME__	De même, se transformera en une chaîne représentant l'heure de traitement du fichier sous le format "hh:mm:ss"
__STDC__	Cette macro n'est censée être définie uniquement que dans les implémentations respectant la norme ANSI. Si c'est le cas, elle doit de plus valoir 1
Attention : les deux caractères _ (*underscore*) font partie du nom de la macro	

2.3. La compilation conditionnelle

Les directives du pré-processeur permettent également de définir des **variables de pré-compilation** qui permettent de réaliser les tests de sélection de parties de fichier source. Ces variables n'ont pas besoin d'être associées à une valeur. Elles jouent en quelque sorte le rôle de booléens puisque le pré-compilateur teste si elles sont définies ou non. Elles servent à déterminer les parties de codes à compiler.

Les variables de pré-compilation se définissent comme suit :

#define *identificateur*

Ainsi, lorsque une variable de pré-compilation a été définie, elle va pouvoir être utilisée pour « sélectionner » des parties de code à l'aide des directives :

#if *expression_constante*
#ifdef *identificateur*
#ifndef *identificateur*
#else
#elif
#endif

Toute condition de sélection de code commence par un ***#if***, un ***#ifdef*** ou un ***#ifndef*** et se termine par ***#endif***, avec éventuellement une partie ***#else***.

Lorsque la condition est vraie, le code qui se trouve entre le ***#if[n[def]]*** et le ***#else*** est passé au compilateur. Si elle est fausse, le code passé est celui entre le ***#else*** et le ***#endif***. Si il n'y a pas de partie ***#else*** aucun code n'est passé.

L'instruction ***#if*** demande en paramètre une expression constante. Notez que l'opérateur **defined** *identifier* retourne la constante 1 si la variable est définie en temps que macro, 0 dans tout autre cas. Pour les deux autres instructions, elles peuvent se définir à partir de la première. Le tableau suivant vous donne les correspondances.

`#if defined identifier`	`#ifdef identifier`
`#if ! defined identifier`	`#ifndef identifier`

Correspondances entre les directives **#if, #ifdef et #ifndef**

Les instructions ***#else*** et ***#elif*** permettent de spécifier du code quand le test n'est pas validé. La différence, entre les deux instructions, tient dans le fait que ***#elif*** permet de spécifier un nouveau test qui, s'il est vérifié, sélectionnera une nouvelle partie de code.

Il peut y avoir autant de ***#elif*** que vous le souhaitez, le ***#else*** ne pouvant apparaître qu'une seule fois. Par contre, seule une condition vérifiée sera acceptée, même si par la suite d'autres conditions pourraient l'être.

Le premier intérêt de la sélection de code est de ne pas inclure de nouveau un fichier déjà inclus par ailleurs. En effet, si le fichier "Titi.h" de l'exemple suivant a déjà été inclus, la variable de pré-compilation *Titi_h* sera définie et toutes les instructions jusqu'au #endif ne seront pas prises en compte. Par contre, si ce fichier n'a jamais été inclus, la variable de pré-compilation *Titi_h* n'existera pas. Elle sera alors définie et les instructions jusqu'au #endif seront prises en compte.

Titi.h

```
#ifndef Titi_h
#define Titi_h
/* texte à n'inclure qu'une seule fois */
#include ''Toto.h''
…
#endif
```

Toto.h

```
#ifndef Toto_h
#define Toto_h

….

#endif
```

Toto.c

```
#include ''Titi.h''
#include ''Toto.h''
```

La sélection est aussi utilisée pour avoir un fichier source contenant des instructions qui donnent des informations sur les variables (traces), et pouvoir générer le fichier exécutable avec ou sans ces traces.

Debug.h

```
#if DEBUG
#define debug(x) printf (‘‘Valeur de #x : %d \n ’’, x)
#else
#define debug(x)
#endif
```

Toto.c

```
#define DEBUG true
#include ‘‘debug.h’’
…
debug (x);
…
```

Et enfon, la sélection de code permet de générer à partir d'un même fichier source des fichiers exécutables pour des machines et des systèmes différents. Le principe de base consiste à passer ou à supprimer des parties de code suivant des conditions fixées à partir des variables de pré-compilation.
Ceci permet d'avoir un même fichier source destiné à être compilé sur plusieurs machines, ou plusieurs systèmes, en tenant compte des différences entre ces machines.
Par exemple :

Sur la machine M1 □ f1(x)
Sur la machine M2 □ f2(x/2)

Cible.h

```
#ifdef M1
#define mafonction  f1(x)
#endif

#ifdef M2
#define mafunction f2(x/2)
#endif
```

Essai.c

```
#define M1
#include ‘‘Cible.h’’
..
```

3. Pré-compilation et enchaîneur de passes

On a vu que les directives du pré-processeur reposent pour beaucoup sur la définition ou non des constantes de compilation et des variables de pré-compilation.
Or pour définir de telles valeur, il faut ajouter au fichier source les directives #define.

Afin d'éviter de modifier systématiquement un fichier dans lequel on a besoin de définir une variable de pré-compilation, on pourra utiliser lors de l'exécution de la compilation, l'option **-D*nom*** avec le nom de la variable de pré-compilation. Le fichier sera alors traité par le pré-processeur comme si la variable de pré-compilation ***nom*** était définie, par la directive ***#define nom***.

De la même manière, pour les constantes de compilation, on peut utiliser l'option **-D*nom*=*valeur*** avec le nom de la constante de pré-compilation suivi du signe "=" et de la valeur à associer à cette constante de pré-compilation. Le pré-processeur changera les occurrences de la variable ***nom*** par la valeur ***valeur***, comme si la constante de pré-compilation nom était définie, par la directive ***#define nom valeur***.

Il existe des variables de pré-compilation qui sont prédéfinies. Elles sont associées au type de la machine, au compilateur et à l'environnement.

Donc, si on considère par exemple la compilation dépendante d'une machine cible (voir exemple ci-dessus), on pourra lorsque l'on exécute le code sur la machine M1 utiliser la commande de compilation suivante :

gcc -DM1 essai.c –o essai

et si on l'exécute sur la machine M2, compiler comme suit :

gcc -DM2 essai.c –o essai

Il est également possible d'effacer ces définitions au niveau de l'appel à l'enchaîneur de passes (gcc en l'occurrence), en utilisant l'option **–U*nom***.

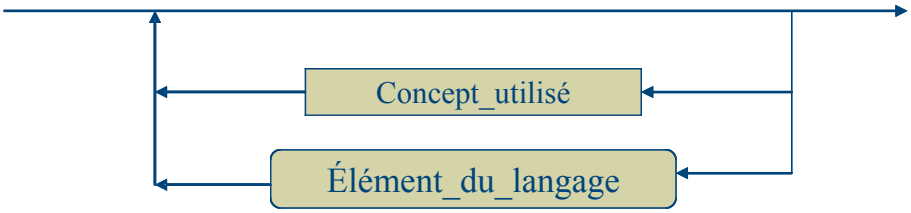
Pour visualiser le résultat d'une pré-compilation, on pourra utiliser l'option –E de l'enchaîneur de passes.
Vu que le nombre de lignes produites par le pré-compilateur peut devenir vite très important (notamment avec l'inclusion des bibliothèques standard), l'idéal est de rediriger la sortie vers un fichier plutôt que de laisser défiler à l'écran.

gcc –E essai.c > precompil_essai.txt

Annexe 2 : Grammaire Générale du C

1. Conventions de notations

Nous utiliserons les Diagrammes de CONWAY décrit ci-dessous :



Et nous utiliserons la Notation littérale ou BNF, à savoir :

- Les concepts utilisés sont écrits en italique
- Les éléments du langage sont écrit en caractères gras
- [ ] représente une partie optionnelle
- + indique une fois ou plus
- * indique qui peut être absent ou répété une fois ou plus
- | indique un ou
- ( ) indique un regroupement

Exemple : description d'un nombre entier sans signe

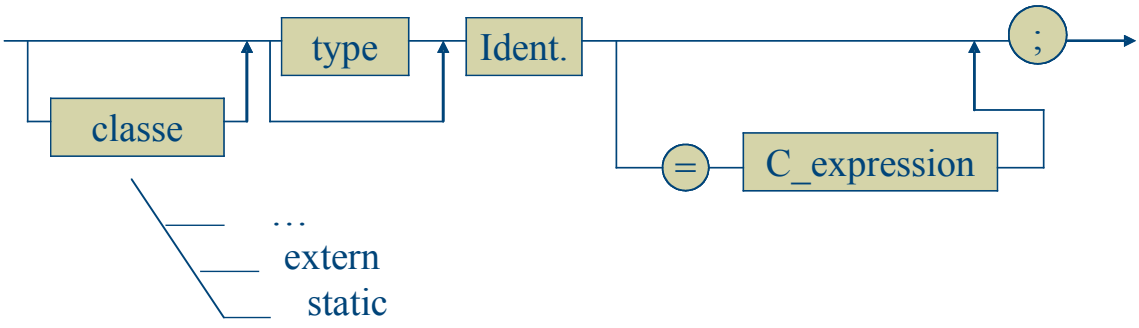
Entier ::= [chiffre]+

Réel ::= entier [. chiffre] +

Chiffre ::= 0|1|2|3|4|5|6|7|8|9

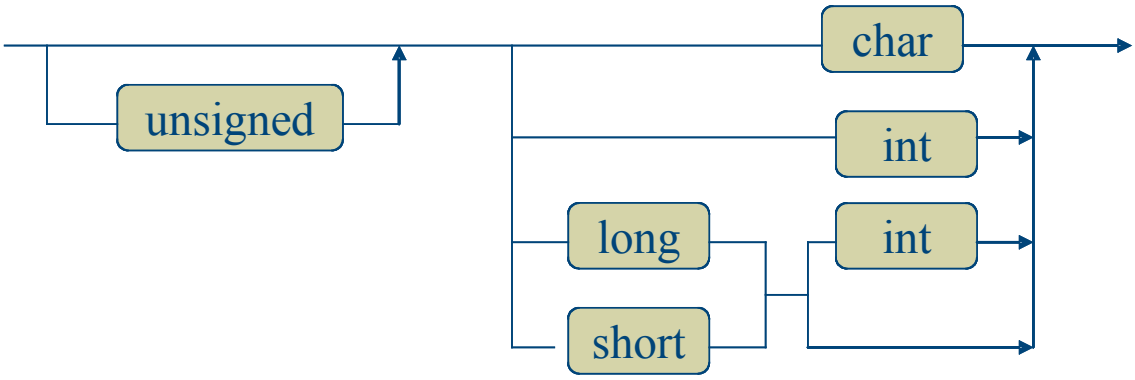
2. Les types et opérateurs de base

Déclaration de ***variable*** :

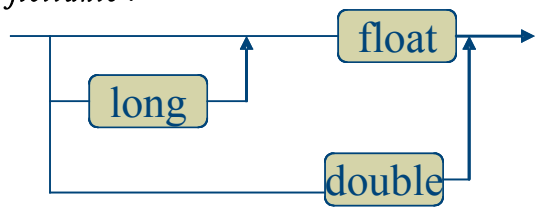


Les *types simples* :

entiers :

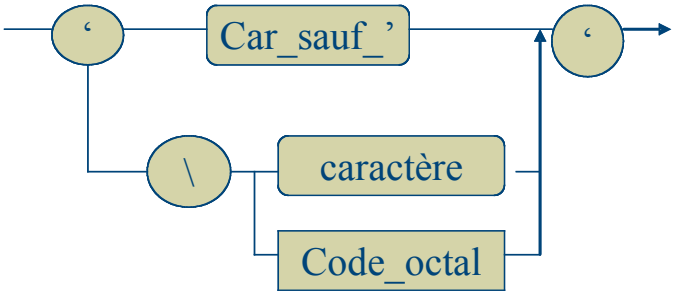


flottants :



Les *constantes* :

caractères :

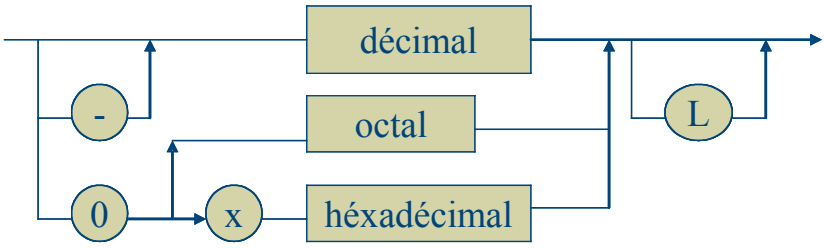


‘x’ ‘X’

‘\’’ ‘\\’ ‘\n’

‘\0’ ‘\7’ ‘\33’

entiers :

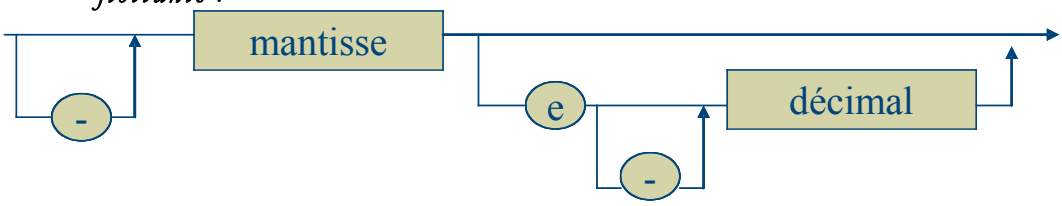


123 -12L

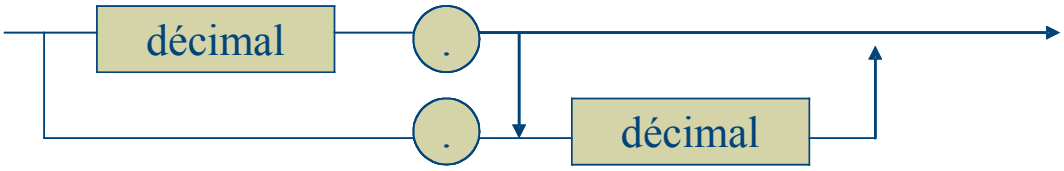
0377 0377L

0xFF 0XAB

flottants :



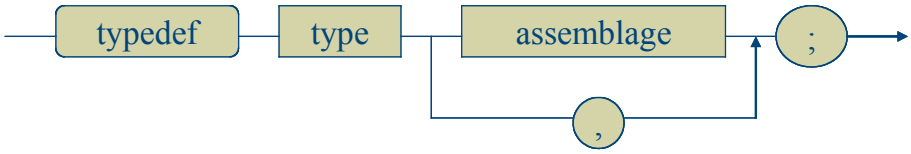
la ***mantisse*** :



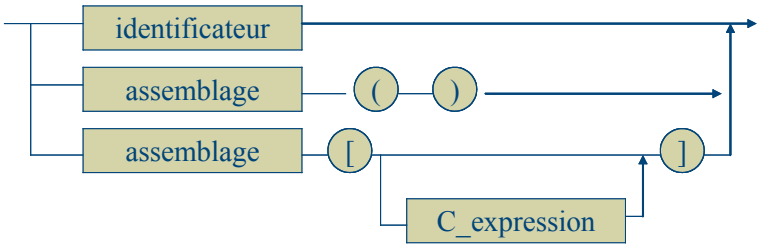
conversion explicite de type :

avec op_conversion :

La ***définition de types*** :

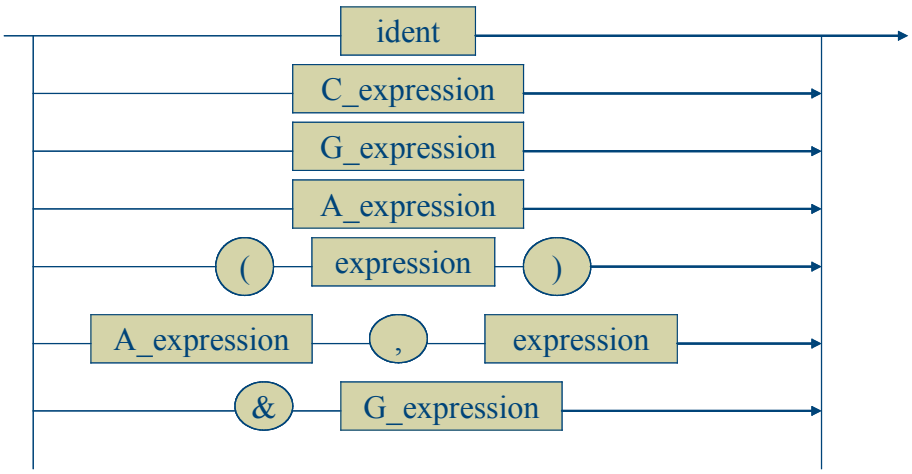


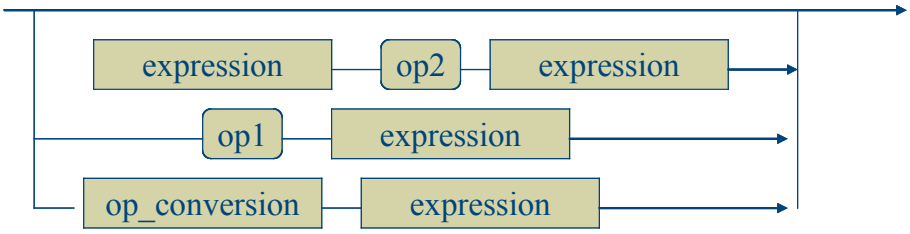
avec ***assemblage*** :



3. Les expressions

Une ***expression*** possède une valeur et peut syntaxiquement s'écrire comme suit :

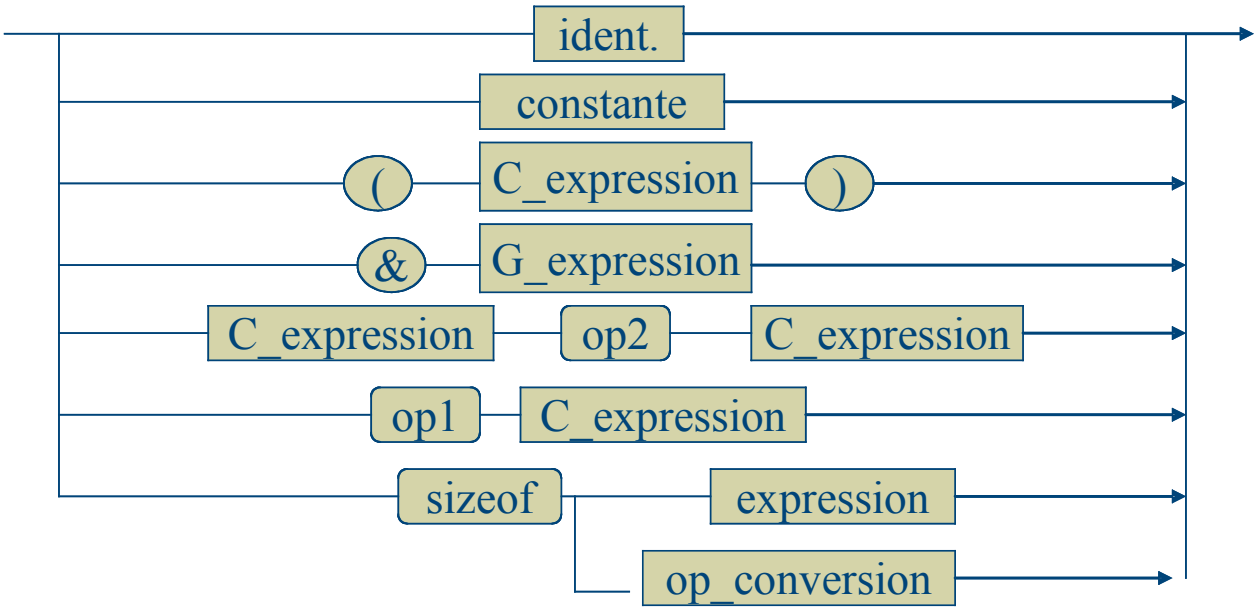




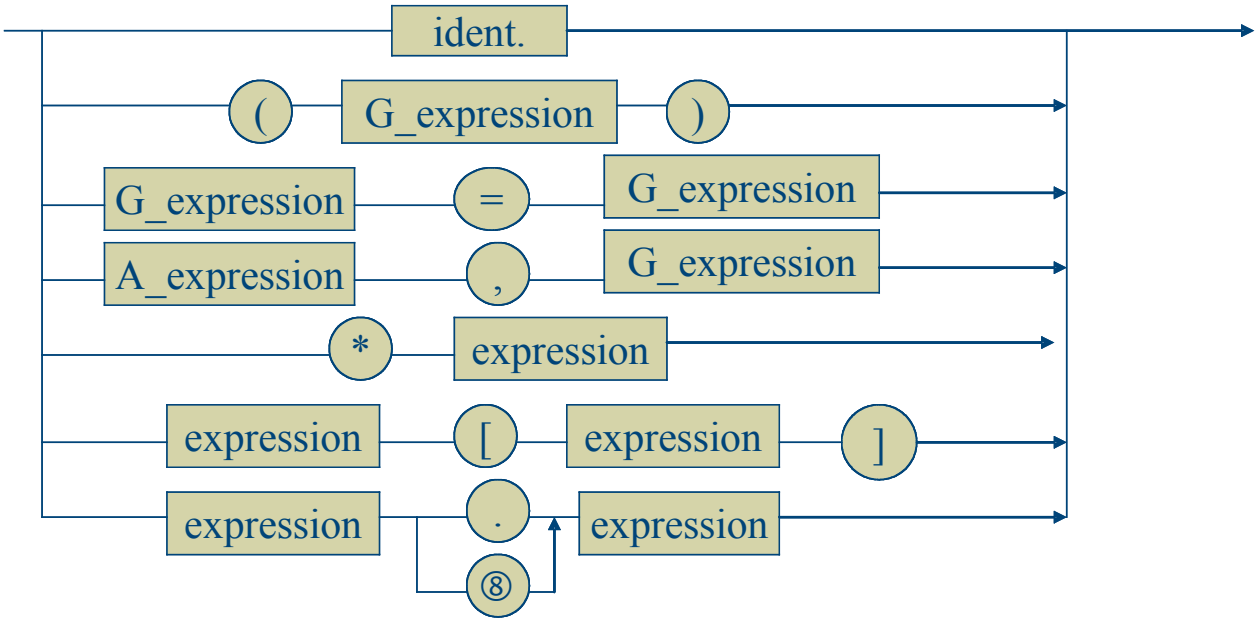
avec :

- Op1 ::= - | ! | ~
- Op2 ::= + | - | * | / | % | << | >> | < | <= | == | >= | > | & | | | ^ | && | || | ^^

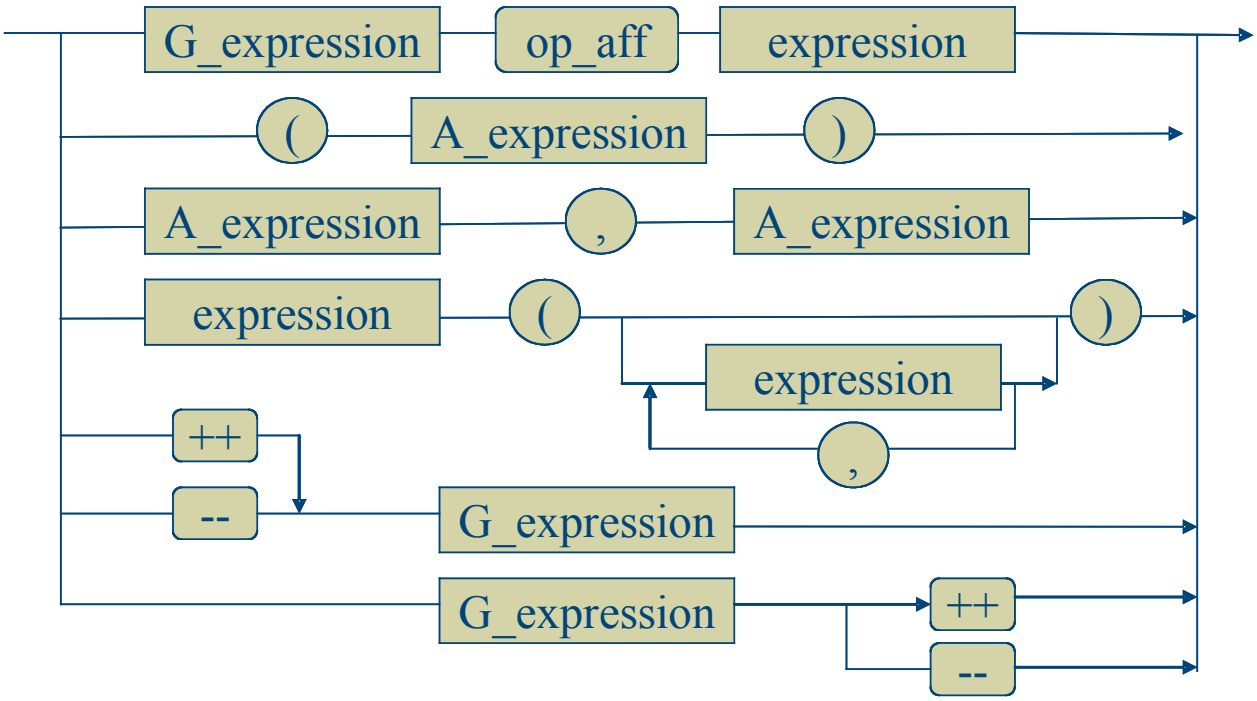
Une *C_expression* est évaluée par le compilateur :



Une *G_expression* correspond à une adresse mémoire :



Une *A_expression* correspond à une action :



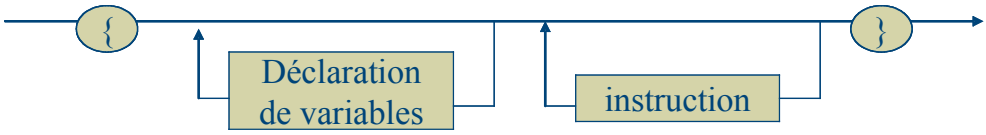
avec :

op_aff ::= = | += | *= | /= | %= | &= | |= | ^= | <<= | >>=

4. Les instructions

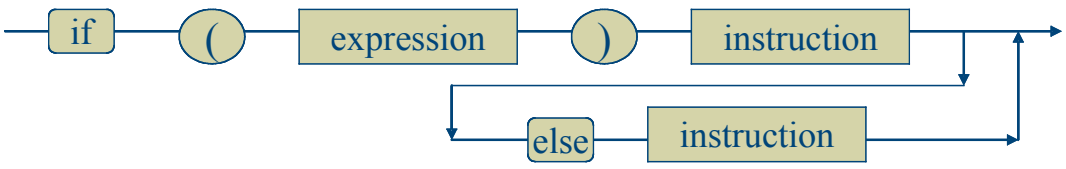
Les *instructions simples* ou action :

Les *blocs* :

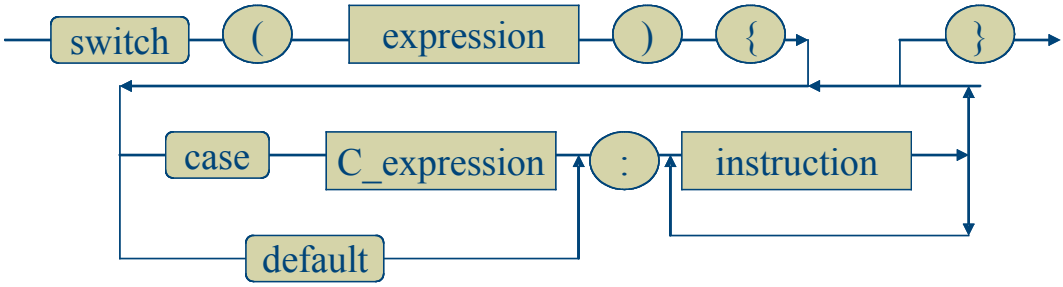


Les *alternatives* :

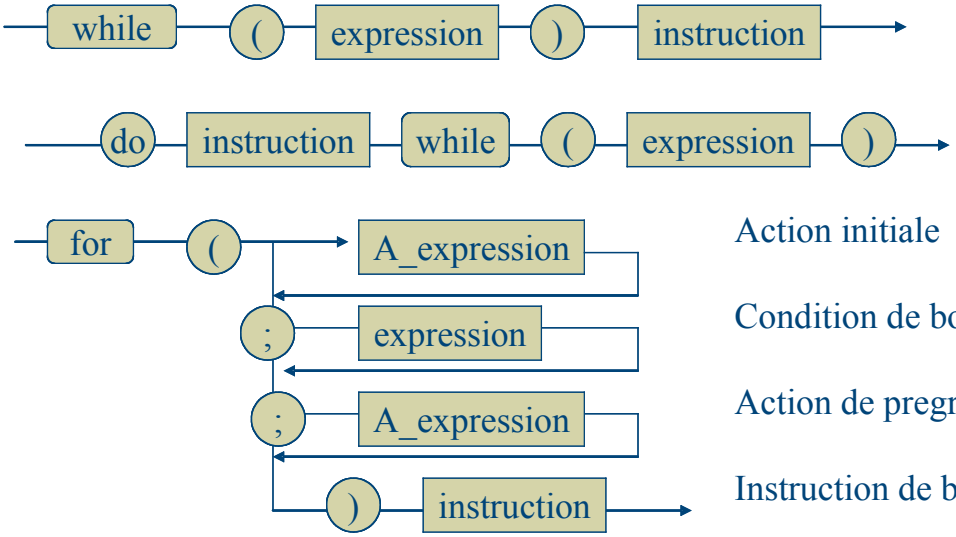
Le *test* :



la ***table de branchement*** (ou « selon ») :



Les ***itératives*** :



Les ***ruptures*** :