Eléments d'Informatique

Cours 2- Programme. Expressions

Catherine Recanati

UNIVERSITÉ PARIS 13 NORD

Plan général

- **Représentation des nombres. Notion de variable.**
- **Programme. Expressions.**
- Architecture des ordinateurs: langage machine, langage assembleur, AMIL.
- Systèmes d'exploitation : fichiers, processus, compilation.
- Instructions de contrôle: boucles et branchements.
- Programme, définition de fonction, appel fonctionnel.
- Tableaux de variables et fonctions d'arguments de type tableau.
- Sens d'un programme, pile d'exécution, compilation.
- Pointeurs et tableaux.
- Chaines de caractères, bibliothèque <string.h>.
- Allocation dynamique, liste chaînées.
- Révisions.

- Cours 2 -
Programme. Expressions.

- Structure d'un Programme C
- Les expressions
- Expressions arithmétiques
- Expressions logiques
- L'instruction if

Plan

Syntaxe

Un programme est écrit dans un langage de programmation qui doit suivre des règles de syntaxe. Il y a des règles pour la formation des « mots », et des règles pour la formation des « phrases ».

Ces règles de grammaire constituent la syntaxe suivie par le langage. Le compilateur ne pourra générer un code machine que si le texte du programme est « bien formé » (*well-formed*), c'est-à-dire syntaxiquement correct.

Plan

Syntaxe

Dans ce cours, nous allons présenter la syntaxe (et la sémantique, c'est-à-dire la signification ou le sens) des différentes tournures du langage C.

Nous le ferons souvent de manière informelle, mais nous donnerons parfois aussi une définition précise de la forme de certaines tournures en utilisant la notation de Bachus Naur (BNF) – comme nous l'avons fait pour les identificateurs de variables dans le cours précédent.

Plan

Structure d'un programme

Un programme C est une suite de déclarations de variables et de fonctions. Il peut aussi contenir des définitions de types et de constantes.

Lors de l'invocation du programme, une fonction particulière est appelée : la fonction `main`. Cette fonction fournit les instructions qui seront exécutées par le programme. Elle fera généralement appel à des fonctions auxiliaires (définies dans le programme ou dans des bibliothèques).

Le langage C propose une librairie standard de fonctions **`stdlib`** qui implémente des opérations courantes et contient des déclarations de constantes et de types d'utilité générale.

Il existe d'autres bibliothèques de fonctions, comme la librairie standard **`stdio`** d'entrées/sorties, ou la librairie de fonctions mathématiques **`math`**.

Pour utiliser les fonctions d'une librairie, il faut inclure le fichier de leurs déclarations, ex: **`#include <stdlib.h>`**

Plan


```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main ()
5   {
6     printf("Bonjour !");
7     return EXIT_SUCCESS;
8   }
```

Ce programme écrit Bonjour ! sur la console et retourne une valeur indiquant une sortie avec succès.

La définition d'une fonction est donnée dans le corps de la fonction qui est un bloc d'instructions entouré d'accolades - les instructions étant séparées par des points-virgules :

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main ()
5   {
6     printf("Bonjour !") ;
7     return EXIT_SUCCESS ;
8   }
```

La syntaxe d'un bloc d'instructions est la suivante :

bloc-instr ::= { seq-instr }

seq-instr ::= (instr ; | decl-var-loc
| def-struct | def-type)*

Cette syntaxe autorise à mélanger des déclarations et définitions à des instructions exécutables, mais on veillera pour plus de clarté, à **bien regrouper en début de bloc toutes les déclarations/définitions, et lister ensuite seulement les instructions.**

Plan

bloc-instr ::= { seq-instr }

seq-instr ::= (instr ; | decl-var-loc
| def-struct | def-type)*

instr ::= instr-base | expr | return [expr]

L'instruction `return` a pour effet d'interrompre la fonction en cours d'exécution et de renvoyer la valeur de l'expression spécifiée si elle existe.

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int x ;    /* variable globale */
5   int main ()
6   {
7     printf("valeur de x: %d",x);
8     return EXIT_SUCCESS;
9   }
--------
Ce programme peut écrire par exemple
     valeur de x: 32767
sur la console.
```

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int x = 0;/* initialisation */
5   int main ()
6   {
7     printf("valeur de x: %d",x);
8     return EXIT_SUCCESS;
9   }
--------
Ce programme écrira cette fois
     valeur de x: 0
sur la console.
```

```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int main (){
5      int x = 0 ; /* x est locale */
6
7      printf("valeur de x: %d",x);
8      return EXIT_SUCCESS;
9    }
--------
```

Cette fois la variable x n'existe que dans la fonction `main`. On dit qu'elle est « locale » à la fonction.

Structure d'un programme C

```
        /* inclusion des fichiers d'en-tête de librairies */
#include <stdlib.h>
        /* déclaration des constantes et des types utilisateur */
        /* déclaration des fonctions */
        /* déclaration des variables globales */

/* définition de la fonction principale appelée main */
int main() {
      /* déclarations des variables locales  */
      /* corps de définition de la fonction main */
}
/* définition des fonctions */
```

Déclaration/définition de fonction

Une **déclaration de fonction** indique le nom de la fonction, le type de la valeur retournée par la fonction, et le type des arguments utilisés lors d'un appel de la fonction.

```
int somme (int x, int y)
```

indique le nom de la fonction: `somme`; le type `int` de la valeur de retour, et le type des arguments formels `x` et `y` (tous les deux `int`).

```
void imprime()
```

indique le nom de la fonction: `imprime`. Le type `void` est utilisé ici pour indiquer qu'il n'y a pas de valeur de retour. L'absence de paramètres formels indique que cette fonction ne prend pas d'arguments.

Déclaration/définition de fonction

Une **définition de fonction** est constituée d'une déclaration de la fonction suivie **du corps de la définition** qui est un bloc d'instructions pouvant faire référence aux paramètres formels.

```
1  int somme (int x, int y)
2  {
3     return (x + y) ;
4  }
```

Structure d'un programme C

```
        /* inclusion des fichiers d'en-tête de librairies */
#include <stdlib.h>
        /* déclaration des constantes et des types utilisateur */
        /* déclaration des fonctions auxiliaires */
        /* déclaration des variables globales */

/* définition de la fonction principale appelée main */
int main() {
      /* déclarations des variables locales  */
      /* corps de la fonction main */
}
/* définition des fonctions auxiliaires */
```

Les expressions

Une **expression** est une instruction exécutable qui représente une valeur.

Il y a plusieurs types d'expressions en C: les affectations, les expressions arithmétiques, les expressions logiques, et d'autres formes que nous introduirons au fur et à mesure.

expr ::= constante | identificateur | (expr)
| affectation | expr-arith | expr-logic
| ...

Les expressions constantes

Constantes entières

- décimale: `372`
- hexadécimale: `0x5a2b`, `0X5a2b` ou `0X5A2B`
- octale: `0477` `010` (= 8 et non pas 10)

On peut les suffixer avec u ou U, et l ou L, pour indiquer `unsigned` et/ou `long`.

Constantes de type `char`

- 'a', '0', ou '\007', '\0', '\13', etc.
- '\n', '\t' , etc.
- '\'', '\\', '\"' ou '"', '\?' ou '?'

Plan

L'affectation

L'affectation permet d'attribuer une valeur à une variable ou à une case mémoire.

affectation ::= $expr_g$ = expr

ex:
```
angle = 45          x = (y+7)/5
tab[2] = 128        *pt = 100
```

Attention: l'expression à gauche du signe = ($expr_g$) doit désigner une case (ou adresse) mémoire. Ce sera une variable, la n^{eme} case d'un tableau, ou la valeur d'un pointeur.

Ainsi (4+ 3) = 2 n'est pas valide.

L'affectation

Après avoir déclaré une variable, on peut lui affecter une valeur pour l'initialiser :

```
1  int x ;
2  int y ;
3  x = 456;
4  y = 0;
5  printf( "-------") ;
6  printf("\n") ;
7  printf( "Valeur de x : %d\n »,x);
8  printf("x= %d et y= %d\n", x, y);
```

Plan


```
1   #include <stdlib.h>
2   #include <stdio.h>
3
4   int x = 4;/* variable globale */
5   int main ()
6   {
7      int x;    /* variable locale */
8      x = 3457;
9      printf("valeur de x: %d\n",x);
10     printf("en hexa: %x\n",x);
11     return EXIT_SUCCESS;
12  }
```

Plan


```
% gcc prog.c
% a.out
valeur de x  : 3457
en hexa  : d81
%
```

Expressions arithmétiques

Les expressions arithmétiques suivent la syntaxe suivante:

expr-arith ::= expr op-binaire expr

| op-unaire expr

op-binaire ::= + | - | * | / | %

op-unaire :: = -

ex:

```
3 + angle*7
8 % 3
-(3*16)
```

Expressions logiques

Le C n'a pas de type booléen explicite.

A la place, une valeur entière est interprétée comme une valeur de vérité :

- si une expression a comme valeur `0`, elle aura comme valeur booléenne *false* (FAUX).
- si une expression a une valeur différente de `0`, elle aura comme valeur booléenne *true* (VRAI).

Expressions logiques

Les expressions booléennes peuvent se combiner avec les opérateurs logiques « et » et « ou ». La table suivante indique les valeurs de vérité de leurs combinaisons:

P	Q	P et Q	P ou Q
F	F	F	F
F	V	F	V
V	F	F	V
V	V	V	V

On peut aussi leur appliquer l'opérateur de négation qui inverse leur valeur.

Expressions logiques

Les expressions logiques suivent la syntaxe suivante:

```
expr-logic  ::= expr op-bin-log expr
              | op-un-log expr
              | expr op-comp-log expr
op-bin-log ::= && | ||
op-un-log  ::= !
op-comp-log ::=  > | < | >= | <=
                 | == | !=

ex: x > 0          (x >=0) && (x < 100)
    !(x>100)       (x > 0) || (x < -2)
    !( (x >=0) && (x < 100) )        !x
```

L'instruction `if`

L'instruction `if` fait partie des instructions dites « de contrôle » qui n'ont pas de valeur (contrairement aux expressions) mais qui modifient l'ordre d'exécution des instructions.

if-cond ::= if (expr) instr [else instr]

Le if conditionnel « simple » (sans else) :

```
1     int x =68;
2     if (x%2 == 0)
3           return x/2;
4    return (x+1)/2;
```

L'instruction `if`

Et le « if-else » :

```
1       int x =68;
2       if (x%2 == 0)
3               x = x/2;
4       else
5               x = (x+1)/2;
6       return x;
```

Plan

L'instruction `if`

```
1     x = 0xffabcd;
2     if (x%2 == 0)
3           x = x/2;
4     else
5     {
6           x = x + 1;
7           return x/2;
8     }
9     return x;
```

Indentation du `else`

```
1     x = 0xffabcd;
2     if (x%2 == 0)
3           x = x/2;
4     else
5     {
6           x = x + 1;
7           return x/2;
8     }
9     return x;
```

Indentation du `else`

```
1     int x =68;
2     if (x%2 == 0)
3           x = x/2;
4     else {
5           x = x + 1;
6           return x/2;
7     }
8     return x;
```

Plan

Merci pour votre attention.

Des questions ?

Les opérateurs utilisés pour décrire les constructions dans le format BNF sont :

– (a) : les parenthèses permettent de délimiter une expression BNF à laquelle un opérateur peut s'appliquer.

– a* : l'étoile représente la répétition (d'une construction a) un certain nombre de fois, éventuellement zéro.

– a+ : le symbole + représente la répétition(d'une construction a) un certain nombre de fois, dont au moins une.

– a | b : représente soit une construction décrite par a, soit une construction décrite par b (c'est soit a, soit b).

– [a] : les crochets indiquent que la présence de la construction a est optionnelle (a apparaît 0 ou 1 fois).

où a , b sont des descriptions BNF de constructions possibles

Les symboles terminaux sont représentés dans une typographie particulière. On a utilisé ici des caractères gras et de couleur rouge, (comme **2**) dans les diapos précédentes.