Les instructions de condition en C

Jean-Gabriel Luque

September 13, 2024

1 Rôle des instructions conditionnelles

Considérons le code suivant

```
...
1 int a, b;
2 assert(scanf("%d%d",&a,&b) == 2);
3 assert(b != 0);
4 printf("%d",a/b);
```

La suite des instructions exécutées sera toujours la même quelque soit les valeurs de `a` et de `b` saisies, sauf dans le cas d'une erreur de saisie ou de la nullité de `b`. Dans ce dernier cas, seules les premieres instructions seront exécutées. L'ordre d'exécution des lignes sera toujours (un préfixe de) `1->2->3->4`.
Avec les outils que l'on a présentés jusqu'ici, on ne peut pas avoir des traces d'exécutions qui dépendent des actions de l'utilisateur.
On aurait pu vouloir, par exemple,

- afficher un message d'erreur si `b == 0`
- proposer un choix d'opérations à effectuer.
- Afficher un message d'erreur et renvoyer `EXIT_FAILURE` en cas d'erreur de saisie
- etc.

Les instructions conditionnelles vont nous permettre de réaliser ce genre de programme.

2 Syntaxes et significations des instructions conditionnelles

2.1 L'instruction `if... else ...+`

Cette instuction permet de réaliser des tests portant sur des expressions booléennes. C'est la plus utilisée des instructions conditionnelles. Sa syntaxe est la suivante

```
if (Expression) {
  séquence d'instructions 1;
} [ else {
  séquence d'instructions 2;
```

```
  }
  ]
séquences d'instructions 3;
```

Les crochets `[...]` sont là pour indiquer que la partie `else` est optionnelle.
Le comportement est le suivant: Si l'évaluation de l'expression située entre parenthèses donne `true` alors le bloc contenant la séquence 1 est exécuté puis le programme reprend sa marche en exécutant la séquence 3.
Si l'évaluation de l'expression située entre parenthèses donne `false` alors bloc contenant la séquence 2 est exécuté (si celui-ci existe) et le programme reprend sa marche en exécutant la séquence 3.

Exemple 1. Considérons le code suivant

```
if (scanf("%d%d", &a, &b) != 2) {
  printf("Erreur de saisie\n");
  return EXIT_FAILURE;
  }

if ( b == 0) {
  printf("Erreur division par 0\n");
} else {
  printf("%d\n", a / b);
}

return EXIT_SUCCESS;
```

Le premier `if` est une alternative à l'utilisation de `assert`. Il permet d'afficher un message d'erreur approprié et de renvoyer une valeur choisie. Si l'appel à `scanf` ne renvoie pas `2` alors le programme affiche un message d'erreur et s'arrête en renvoyant la valeur `EXIT_SUCCESS`. Notez que la présence du mot clef `return` arrête la fonction. Les instructions suivantes ne sont donc pas exécutées. Il n'est donc pas nécessaire d'ajouter un `else`.
Si l'appel à `scanf` renvoie 2 le programme passe à l'instruction suivante: si `b == 0` alors le programme affiche un message d'erreur, sinon il affiche le résultat de l'opération `a / b`. Dans les deux cas, la ligne `return EXIT_SUCCESS;` est exécutée.
Notez que l'on peut considérer le cas `b == 0` comme une erreur et sortir avec `EXIT_FAILURE`. Dans ce cas, on peut omettre le mot clef `else`.

```
if (scanf("%d%d", &a, &b) != 2) {
  printf("Erreur de saisie\n");
  return EXIT_FAILURE;
  }

if ( b == 0) {
  printf("Erreur division par 0\n");
  return EXIT_FAILURE;
}

printf("%d\n", a / b);
```

```
return EXIT_SUCCESS;
```

Notez bien les `{...}` suivant `if` et `else` car ceux-ci signifient que l'on créé des blocs d'instructions. Il est donc important de bien faire attention à la portée des variables.

Exemple 2. Le code

```
1  int a = 0;
2  if (a == 0) {
3    int a = 1;
4    printf("%d\n",a);
5  }
6  printf("%d\n",a);
```

Affiche

```
1
0
```

En effet, la variable modifiée à la ligne (3) est celle du bloc (2)-(5) et non celle déclarée à la ligne (1).

Les blocs d'instructions suivant les `if` et `else` peuvent contenir n'importe quel type d'instructions, y compris d'autres instructions conditionnelles. L'enchevêtrement des blocs peut rendre le programme difficile à lire. Afin de faciliter la lecture, il est demandé de placer toujours `} else {` sur la même colonne que le `if` correspondant et que les accolades fermantes soit sur la même colonne que le `if` ou le `} else {` correspondant. Il est aussi demandé de respecter l'indentation d'usage, en décalant les instructions à l'intérieur d'un même bloc.

Exemple 3. Examinons le code suivant

```
  if (scanf("%d %c %d",&a,&op,&b) != 3) {
    return EXIT_FAILURE;
  }

 if (op == '+') {
    printf("%d %c %d = %d\n", a, op, b, a + b);
  } else {
    if ( op == '-') {
      printf("%d %c %d = %d\n", a, op, b, a - b);
    } else {
      if (op == '*') {
         printf("%d %c %d = %d\n", a, op, b, a * b);
      } else {
        if (op == '/') {
          if (b!= 0) {
            printf("%d %c %d = %d\n", a, op, b, a * b);
          } else {
```

```
            printf("Erreur division par 0\n");
            return EXIT_FAILURE;
          }
        } else {
          printf("Opération inconnue");
          return EXIT_FAILURE;
        }
      }
    }
  }


  return EXIT_SUCCESS;
```

Celui-ci demande une opération à l'utilisateur une opération à effectuer sous la forme `int op int`, `op` désignant le symbole d'une opération. Par exemple, `5 + 5`. Le lecteur curieux s'intéressera au format utilisé dans le `scanf` et en particulier à la présence des espaces.
On voit ici que les blocs sont assez imbriqués et que, lorsqu'on veut taper le programme, on se trompe assez facilement dans les ouvertures et les fermetures des blocs.

Remarque 1. Si `b` est une variable booléenne, il est redondant d'écrire `if (b == true)` ou `if (b == false)` car le tableau de vérité de `b == true` est le même que celui de `b` et le tableau de vérité de `b == false` est le même que celui de `!b`. Il faut donc écrire plutôt `if (b)` ou `if (!b)`.

2.2 L'instruction `switch`

Lorsque le test porte sur des entiers (ou un type apparenté à des entiers), on peut utiliser la commande `switch` pour rendre le programme plus lisible. Sa syntaxe est la suivante

```
switch (expression) {
  case V1:
          Séquence 1;
          [break;]
  case V2:
          Séquence 2;
          [break;]
  ...
  case Vn:
          Séquence n;
          [break;]
  [default:
          Séquence n+1;]
}
```

L'évaluation de l'expression doit donner un type apparenté à un entier (par exemple un `int`, un `char` etc. mais pas un `double`) de même type que les valeur `V1`, $\cdots$, `Vn`. Si l'évaluation donne la valeur `Vk` la séquence k est exécutée. Si la valeur n'apparaît devant aucun des `case` alors la séquence n+1 est exécutée si le cas `default` est présent sinon l'instruction ne fait rien.

Remarque 2. `V1,...,Vn` *peuvent être des expressions donnant une valeur du même type que "*`expression`*".*

Notez la présence des mots clefs `break` notés comme optionnel. Sans eux le programme exécute toutes les séquences qui suivent le cas detecté jusqu'à ce qu'il lise un `break` ou qu'il atteigne la fin du bloc de l'instruction `switch`.

Exemple 4. Le programme de l'exemple 3 peut se réécrire de la façon suivante

```
  if (scanf("%d %c %d",&a,&op,&b) != 3) {
    return EXIT_FAILURE;
  }

  switch (op) {
    case '+':
              printf("%d %c %d = %d\n", a, op, b, a + b);
              break;

    case '-':
              printf("%d %c %d = %d\n", a, op, b, a - b);
              break;

    case '*':
              printf("%d %c %d = %d\n", a, op, b, a * b);
              break;
    case '/':
              if (b != 0) {
                printf("%d %c %d = %d\n", a, op, b, a * b);
              } else {
                printf("Erreur division par 0\n");
                return EXIT_FAILURE;
              }
              break;
    default:
              printf("Opération inconnue");
              return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

Remarque 3. L'instruction `switch` n'apporte rien de plus par rapport à l'instruction `if`. Tout ce qui est faisable avec une instruction `switch` est réalisable avec des `if` et réciproquement!
Il s'agit donc de choisir l'instruction à utiliser sur des critères de lisibilité et d'esthétique.

Remarque 4. L'omission du mot clef `break` est utile lorsque l'on veut que plusieurs valeurs provoquent l'exécution de la même séquence. Dans l'exemple 4, si on veut que l'on puisse utiliser les symboles `x` ou `*` pour désigner la même opération, on peut écrire

```
...
    case 'x':
    case '*':
              printf("%d %c %d = %d\n", a, op, b, a * b);
              break;
...
```

2.3 L'opérateur conditionnel `?:`

L'opérateur conditionnel permet de conditionner la valeur d'une expression. Sa syntaxe est la suivante

```
( Condition ) ? Exp1 : Exp2
```

`Condition` est une expression booléenne. Si son évaluation vaut `true` alors l'expression `Exp1` sera évaluée, sinon ce sera l'expression `Exp2`.
Par exemple:

```
c =  (b != 0 ? a / b : 0);
```

placera dans la variable `c` la valeur de `a / b` si `b != 0` et `0` sinon.
Cet opérateur permet de réduire la taille des programmes et doit être utiliser pour contribuer à la clarté du code.

3 Structuration des conditions

Nous ne rappelerons jamais assez que l'écriture d'un programme commence avec du papier et un crayon. Il faut tout d'abord bien analyser le problème et traiter des exemples à la main. Une fois que l'on a trouvé la méthode que l'on va implanter, il faut la formaliser afin d'avoir une vue d'ensemble sur le problème et la solution que l'on propose.
En particulier, il faut visualiser la structure des conditions qui apparaîtront dans le programme. Il est fortement conseillé de les dessiner sous forme d'un arbre AVANT de se lancer dans la programmation afin

- de ne pas oublier de cas et ne pas avoir de redondance,
- de bien choisir les instructions conditionnelles que l'on va utiliser,
- d'indenter correctement le programme,
- de simplifier le plus possible les tests dans le but de rendre le programme le plus lisible possible.

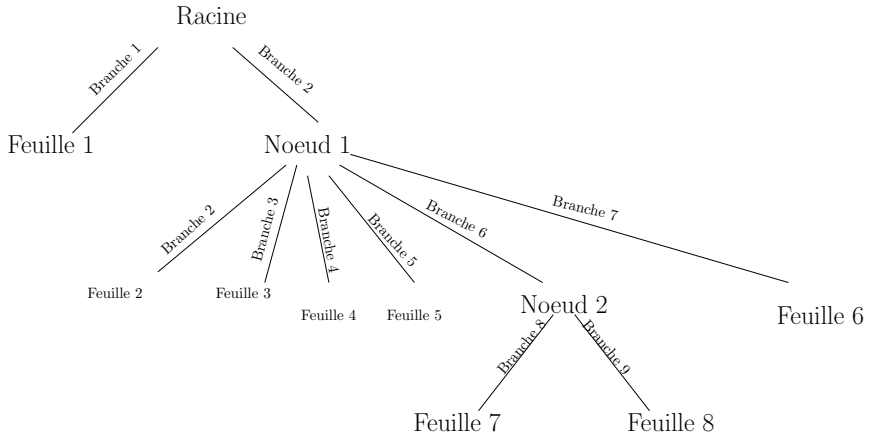
Pour visualiser la structure des tests, nous utiliserons la notion d'arbre. Un arbre est

- un dessin composé d'une racine, de branches, de noeuds et de feuilles,
- les feuilles sont des noeuds terminaux et la racine le noeud initial,
- la racine est le point de départ de la lecture,

- à partir de chaque noeud, on peut lire une branche afin de se rendre à un autre noeud ou à une feuille,
- les branches et les noeuds sont disposés de façon à ce qu'aucune lecture circulaire ne soit autorisé,
- un chemin valide est la lecture d'une suite de noeuds et de branches menant de la racine jusqu'à une feuille.

Les arbres que l'on dessinera sont destinés à représenter la structure de test d'un programme. Afin de faciliter leur lecture, nous les dessinerons à l'envers, c'est à dire la racine en haut et les feuilles en bas. De cette façon, la lecture se fera dans le même sens que celle du programme.

Exemple 5. Le dessin suivant est un arbre:



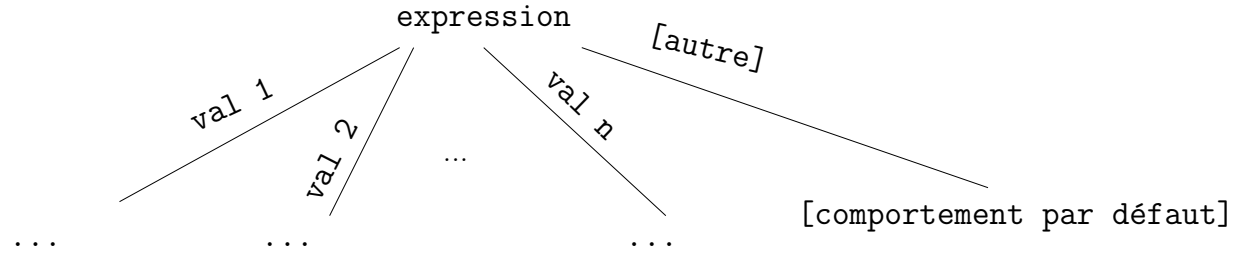
Le chemin

```
Racine -Branche 3-> Noeud 1 -Branche 6-> Noeud 2 -Branche 9-> Feuille 8
```

mène de la racine à la feuille 8.

3.1 L'arbre des tests

Un arbre de test sera un arbre dont les noeuds seront étiquetés par des expressions, les branches par des valeurs et les feuilles par des actions. Un arbre symbolise l'enchevêtrement des tests dans un programme et chaque chemin une trace possible.
Un arbre de test doit avoir la forme suivante:

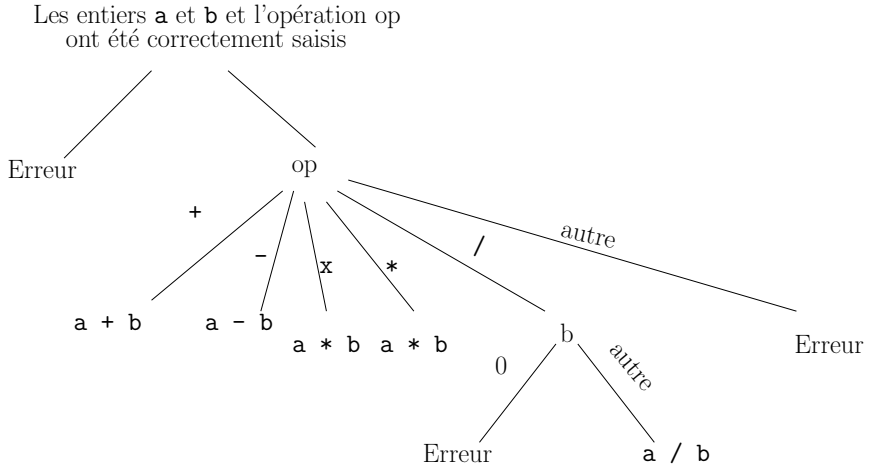


(1)

Il signifie que si la valeur de `expression` est `val k` alors on appliquera l'action `k`. Si cette l'évaluation ne donne aucune des valeurs figurant sur les branches alors on applique le comportement par défaut, qui peut lui aussi être décrit par un (sous-) arbre. Ce dernier est optionnel, si il

est manquant alors aucune action ne sera éxecutée dans les cas non listés dans les étiquettes des branches.

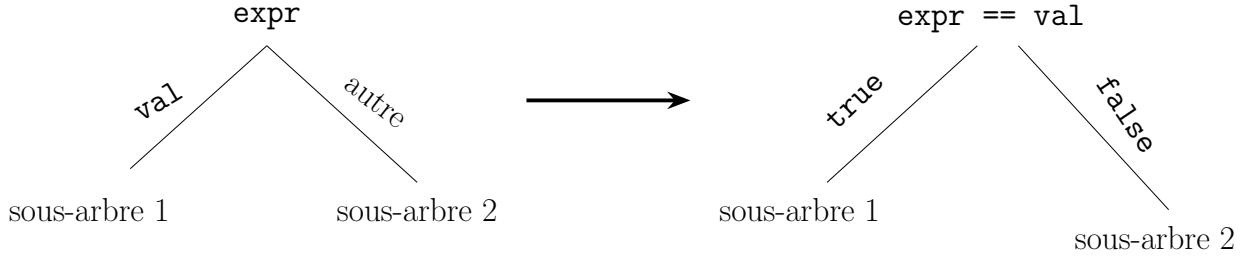
Exemple 6. Nous voulons réaliser un programme qui demande deux entiers à l'utilisateur et un symbole et qui affiche le résultat de l'opération. Une rapide analyse du problème permet de voir que l'on doit suivre l'arbre suivant:



Lorsque l'on passe à la programmation, l'interprétation de l'arbre (1) est simple: il permet d'écrire le code suivant

```
switch (expression) {
  case val 1:
      /* action 1*/
      break;
  case val 2:
      /* action 2*/
      break;
   /*....*/
  case val n:
      /* action n*/
      break;
  default:
  /* comportement par défaut*/
}
```

Attention à ne pas oublier les `break`.
Lorsque l'arbre n'a que deux branches dont l'une est étiquetée `autre`, il est conseillé transformer l'arbre de façon à faire apparaître une expression booléenne:

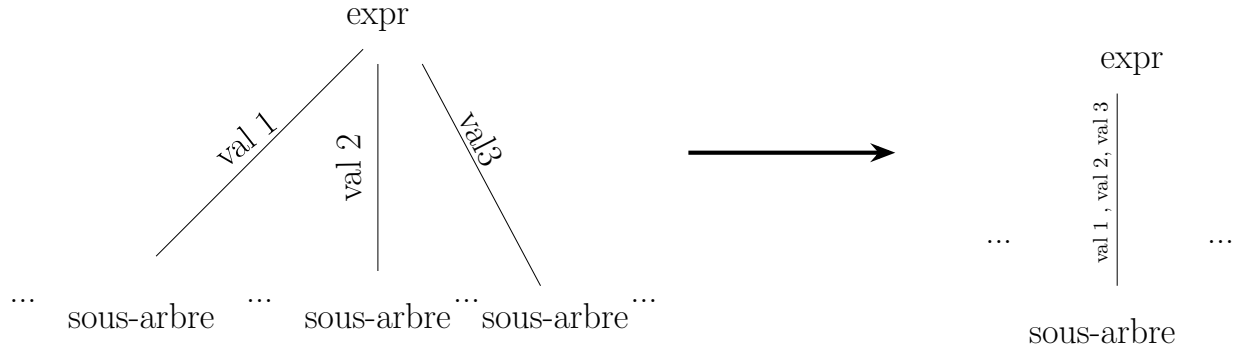


Ce nouvel arbre se traduit par l'instruction

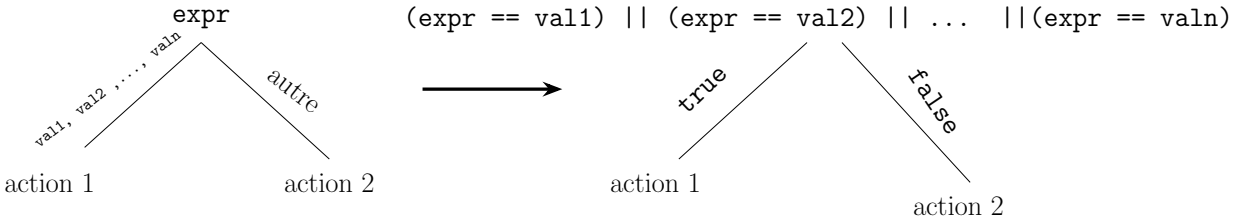
```
if (expr == val) {
  /* action 1*/
} else {
  /* action 2*/
}
```

Remarque 5. S'il n'y a qu'une branche, l'arbre se traduit par un `if` sans `else`.

Si plusieurs branches mènent à la même action, on peut les regrouper:



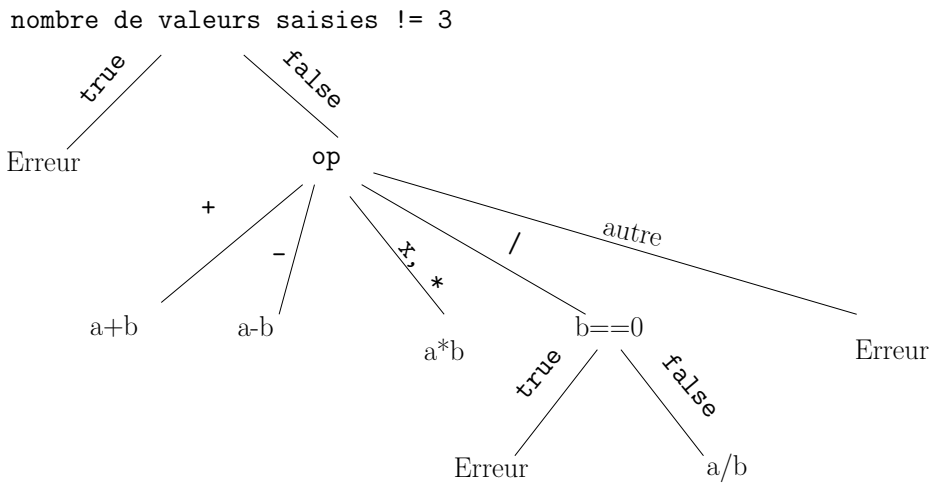
Remarque 6. Si après cette simplification, il n'y a plus qu'une seule branche ou deux branches dont l'une est étiquetée "autre" alors il est conseillé de transformer cet arbre en utilisant une expression booléenne:



Ce nouvel arbre se traduit par l'instruction

```
if (expr == val1 || expr == val2 || ... || expr == valn) {
  /* action 1*/
} else {
  /* action 2*/
}
```

Exemple 7. L'arbre donné dans l'exemple 6 devient:



Cet arbre permet d'obtenir le code donné dans l'exemple 4... Enfin presque, si on appliquait strictement les règles données ci-dessus, le code devrait être

```
if (scanf("%d %c %d", &a, &op, &b) != 3) {
  return EXIT_FAILURE;
} else {
  ....
  }
```

C'est l'utilisation de `return` qui permet de se passer du `else` et de retrouver le code que l'on attendait.

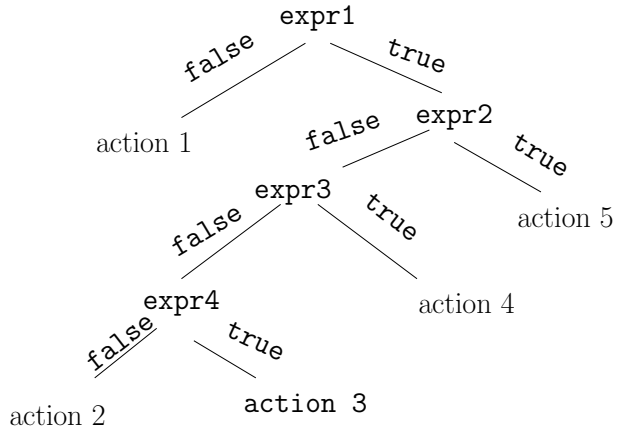
3.2 Arbres de test et logique booléenne

La fin du paragraphe précédent montre qu'il existe souvent plusieurs arbres (et donc plusieurs structures de test... et donc plusieurs codes) permettant de résoudre un même problème. On dira que ces arbres sont équivalents. Nous allons formaliser ici ce que l'on entend par la notion d'équivalence.
L'idée est que pour des conditions similaires, portant sur les variables, dans deux arbres équivalents, les mêmes actions soient exécutées.

Remarque 7. Des arbres équivalents donneront des programmes équivalents dans leurs exécutions. Le choix la structure incombe au programmeur et celui-ci doit prendre en compte des critères de lisibilité. Lorsqu'une action apparaît à plusieurs endroits dans un arbre, il y a la possibilité de les regrouper. En faisant cela, on simplifie l'arbre mais on rend plus complexe les expressions que l'on va tester. Il s'agit alors d'opter pour le bon compromis afin que le programme résultant soit le plus clair possible.

Lorsque l'on suit un chemin de la racine à une feuille, chaque expression rencontrée doit avoir la valeur de la branche que l'on va emprunter. Pour être plus précis, si on considère une chemin de la forme `exp1 -val1-> exp2 -val2->...expn-valn->action`, l'action `action` sera exécutée si `(exp1 == val1) && (exp2 == val2) && ... && (expn == valn)`.

Exemple 8. Considérons un arbre de test sur des expressions booléennes:



L'action `action 3` sera exécutée si
`expr1 == true && expr2 == false && expr3 == false && expr4 == true`
autrement dit si `expr1 && !expr2 && !expr3 && expr4` .

On peut procéder de la même façon avec tous les chemins. Ceci implique que l'on peut proposer un programme composé uniquement d'instructions `if` non imbriquées et sans `else`.

```
if (!expr1) {
  /* action 1*/
}

if (expr1 && !expr2 && !expr3 && ! expr4) {
  /* action 2*/
}

if (expr1 && !expr2 && !expr3 &&  expr4) {
  /* action 3*/
}

if (expr1 && !expr2 && expr3) {
  /* action 4*/
}

if (expr1 && expr2) {
  /* action 5*/
}
```

Remarquons tout de même que, pour des raisons de lisibilité, il n'est pas conseillé de proposer ce genre de programme.

Supposons qu'une même action soit exécutées pour plusieurs chemins différents.
Notons `{chemin_1, chemin_2,...,chemin_n}` l'ensemble des chemins menant à executer cette action. Celle-ci sera exécutée si et seulement si

`(expression associée au chemin_1) || ... || (expression associée au chemin_n).`

Exemple 9. Si on suppose que dans l'exemple 8 l'action 1 et l'action 3 soient les mêmes , on peut les regroupes en utilisant une disjonction (symbole `||` / opérateur "ou logique"). L'action 1 sera executée si et seulement si `!expr1 || (expr1 && !expr2 && !expr3 && expr4)`. Cette expression peut être utilisée dans le programme:

```
if ( !expr1 || (expr1 && !expr2 && !expr3 &&  expr4) ) {
  /* action 1*/
}

if (expr1 && !expr2 && !expr3 && ! expr4) {
  /* action 2*/
}

if (expr1 && !expr2 && expr3) {
  /* action 4*/
}

if (expr1 && expr2) {
  /* action 5*/
}
```

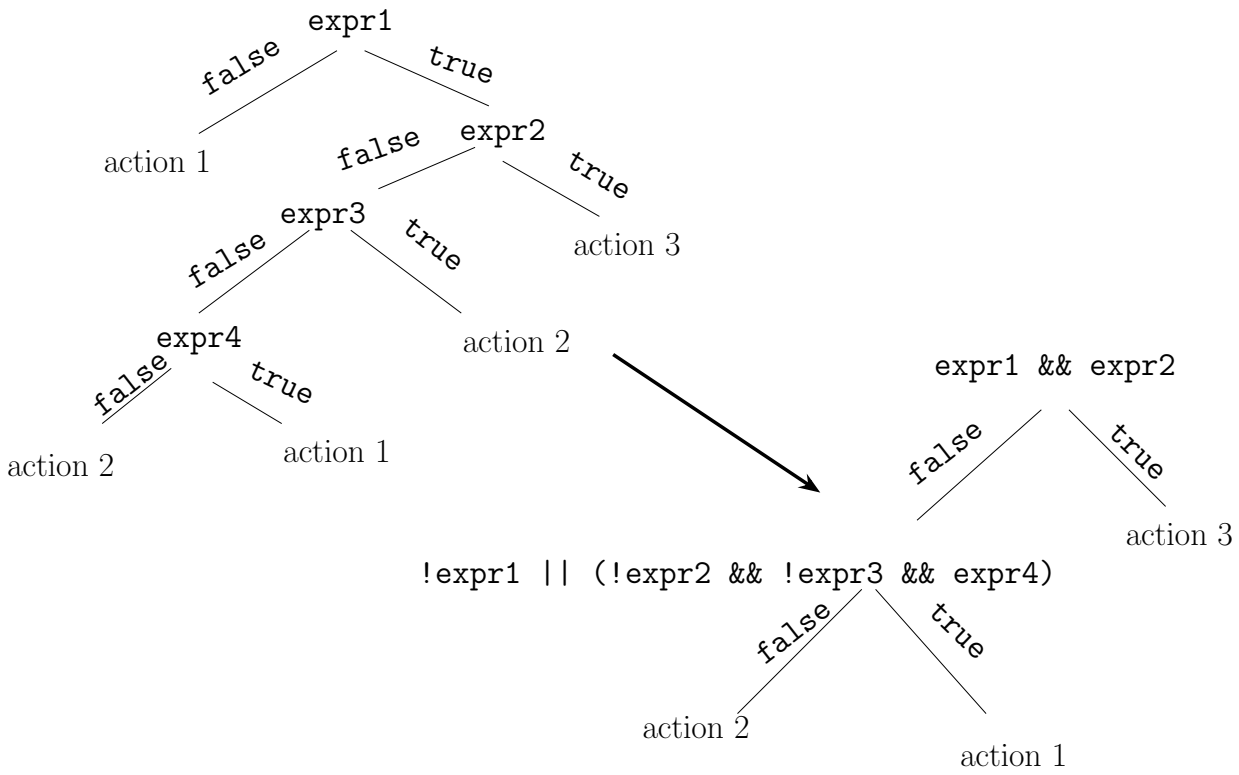
Comme on vient de le voir, la méthode permet d'obtenir un code pour lequel chaque action n'est écrite qu'une seule fois. Néanmoins, cela peut rendre les conditions plus difficiles à lire.
On peut utiliser le "`else`" afin d'améliorer la lisibilité. On peut procéder de la façon suivante:

- On repère dans l'arbre l'action la plus facile à décrire par une expression. Nous la noterons `expr` pour la suite.
- On écrit

  ```
  if (expr) {
    /*action*/
  } else {
  ...
  }
  ```

- Dans la partie `else`, on s'occupe du reste de l'arbre en sachant que la condition `!expr` est satisfaite et que ceci permet des simplifications dans les expressions.

Exemple 10. En suivant cette méthode, on obtient:



L'expression

```
!expr1 || (!expr2 && !expr3 && expr4)
```

provient de la simplification de

```
  (!expr1 || (expr1 && !expr2 && !expr3 && expr4))
```

sachant `!(expr1 && expr2)`.
Le code devient alors

```
if (expr1 && expr2) {
  /*action 3*/
} else {
  if (!expr1 || ( !expr2 && !expr3 && expr4)) {
    /* action 1*/
  } else {
    /* action 2*/
  }
}
```

On peut remarquer que l'on n'a pas besoin de connaître l'expression associée à `action 2`.

Remarque 8. Attention, la méthode de simplification expliquée ci-dessous n'est valable que lorsqu'aucune instruction, entre deux tests consécutifs, ne vient modifier les valeurs des expressions. Par exemple, il ne faut pas l'appliquer au cas suivant:

```
if ( b != 0) {
  b = a % b;
```

```
if ( b == 0) {
  ...
```

car la valeur de `b` a été modifiée entre les deux tests.
Si on veut prendre en compte le cas général, il faut faire appel aux notions de pre-assertion et post-assertion qui seront développées au second semestre.