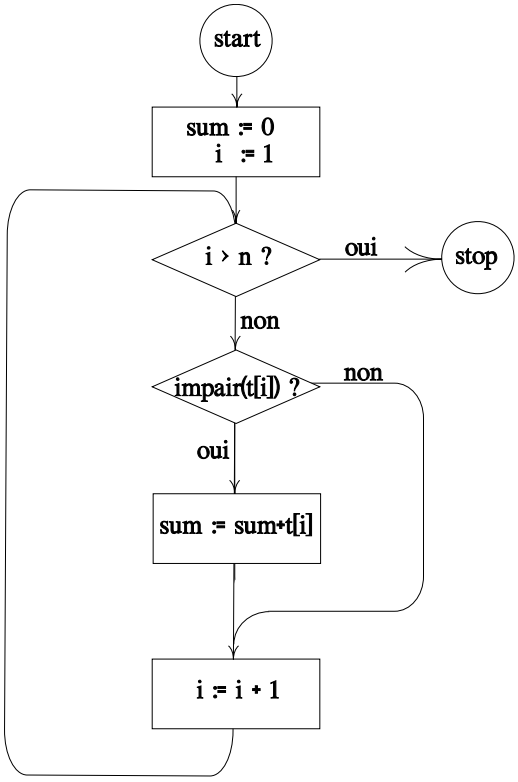
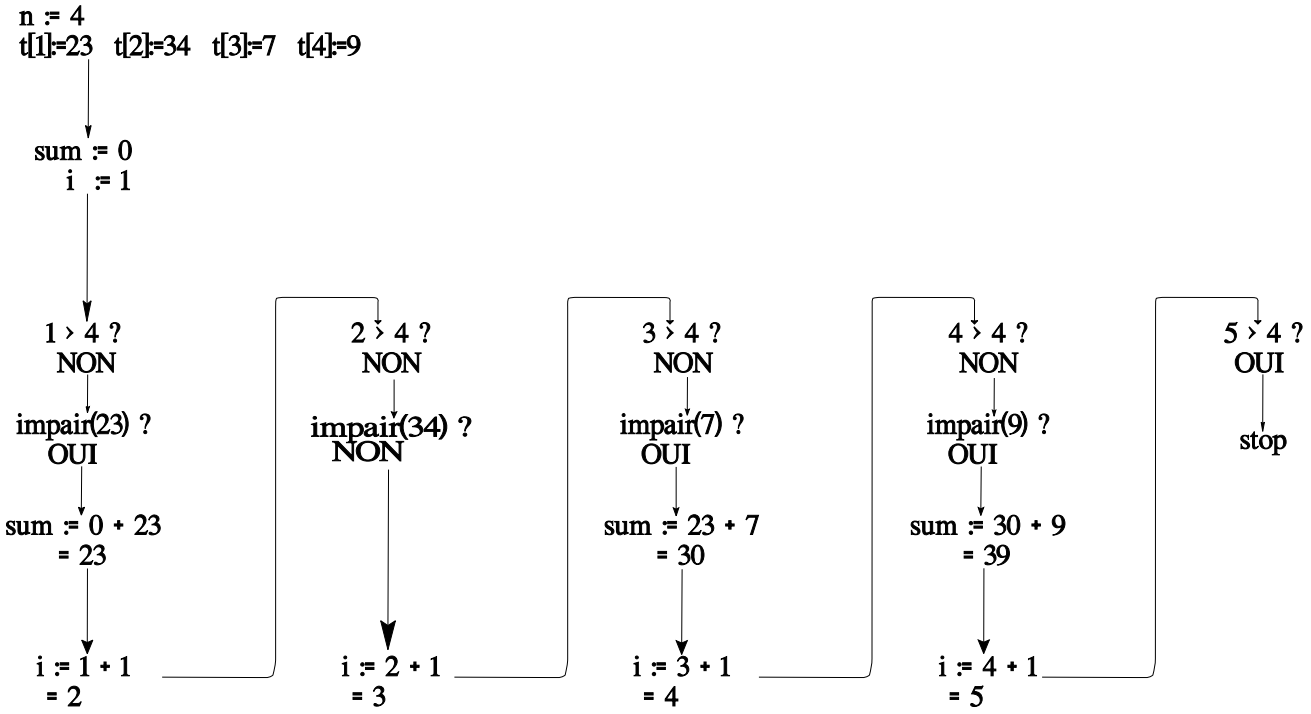
Algorithmes & Programmes

Dans cette partie, nous allons prendre un algorithme simple et le traduire en un ensemble de programmes dans des langages différents. Ceci nous montrera les différences entre divers langages de programmation, concernant la lisibilité des programmes, leur simplicité ou complexité, et, surtout, les manières différentes de penser un programme dans un langage ou un autre.

Ci-dessous une suite de programmes exemples qui calculent tous la somme des nombres impairs d'un ensemble de nombres. Tout d'abord une sorte d'organigramme:



et voici un suivi, pas à pas, du calcul que cet organigramme décrit :



Le programme en langage machine (binaire) Motorola 68000

code machine	commentaires
00100100 01011111	adresse de retour dans A2
00100010 01011111	adresse de premier élé dans A1
00110010 00011111	compteur n dans D1
01000010 01000010	la somme dans D2 initialisé 0
01001110 11111010 00000000 00001110	saut vers fin boucle si n = 0
00001000 00101001 00000000 00000000 00000000 00000001	LOOP: si A1 est pair
01100111 00000010	alors vers SUIVANT
11010100 01010001	sinon d2 ← D2 + A1
01010100 01001001	SUIVANT: A1 prend l'élé suiv.
01010001 11001001 11111111 11110010	décrément D1, si ≠ 0 vers LOOP
00111110 10000010	D2 sur la pile
01001110 11010010	retour vers appelant

et ensuite en langage assembleur Motorola 68000

étiquettes	instructions		commentaires
SUMODD:	MOVE.L	(A7)+,A2	dépile adresse retour dans A2
	MOVE.L	(A7)+,A1	dépile adresse du premier élément dans A1
	MOVE.W	(A7)+,D1	dépile le nombre d'éléments n dans D1
	CLR.W	D2	mettre D2, la somme, à 0
	JMP	COUNT	saut vers la fin de boucle pour tester si n = 0

LOOP:	BTST	0,1(A1)	si l'élément adressé par A1 est pair
	BEQ.S	SUIVANT	alors continuer à SUIVANT
	ADD.W	(A1),D2	sinon ajouter l'élément à D2
SUIVANT:	ADDQ.W	#2,A1	mettre dans A1 l'adresse de l'élément suivant
COUNT:	DBF	D1, LOOP	décrémenter D1; sauf s'il est = -1 continuer en LOOP
	MOVE.W	D2,-(A7)	empiler la somme contenue dans D2
	JMP	(A2)	retourner vers l'adresse contenue dans A2

Ensuite le même programme en un langage légèrement plus évolué : le langage Basic. Ce langage, dont le nom est un acronyme de Beginer's All-purpose Symbolic Instruction Code, a été initialement développé (vers la fin des années 1950, au Dartmouth College sous la direction de John Kemeny) pour faciliter l'apprentissage de la programmation. Entre temps, c'est probablement le langage le plus utilisé sur des micro-ordinateurs personnels. Voici alors le listing du programme :

```
0100 DIM T(100)
0200 READ N
0300 FOR I = 1 TO N
0400   READ T(I)
0500 NEXT I
0600 GOSUB 1100
0700 PRINT S
0800 GOTO 2000
0900 DATA 4
1000 DATA 23, 34, 7, 9
1100 REM S SERA LA SOMME DES ELEMENTS IMPAIRS DE T(I)
1200 LET S = 0
1300 FOR I - 1 TO N
1400    IF NOT ODD(T(I)) THEN GOTO 1600
1500   LET S = S + T(I)
1600 NEXT I
1700 RETURN
2000 END
```

Quelques extraits de ce même programme, cette fois-ci écrit en langage Cobol. Ce langage, acronyme pour COmmon Buisiness Oriented Language, est (encore) très répandu dans des applications informatiques à la gestion d'entreprises. Il a été développé par une équipe sous la direction de Grace Murray Hopper, initialement pour harmoniser les tâches de programmation à l'intérieur de l'armée américaine. Dans ce contexte, Cobol est lentement remplacé par le langage ADA. Notons que

Cobol essaye d'utiliser des instructions et des combinaisons d'instructions ressemblant à l'anglais.

Voici alors une partie du listing de notre programme en langage Cobol :

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUMERIC-VARIABLES USAGE IS COMPUTATIONAL.
  01 TERMS PICTURE 9999 OCCURS 100 TIMES INDEXED BY I.
  02 N PICTURE 999.
  02 SUM PICTURE 999999.
  02 HALF-TERM PICTURE 9999.
  02 RMDR PICTURE 9.

PROCEDURE DIVISION.
EXAMPLE.
  MOVE 23 TO TERMS(1).
  MOVE 34 TO TERMS(2).
  MOVE 7 TO TERMS(3).
   MOVE 9 TO TERMS(4).
   MOVE 4 TO N.
 PERFORM SUM-ODDS.

SUM-ODDS.
 MOVE 0 TO SUM.
 PERFORM TAKE-ONE-TERM VARYING I FROM 1 BY 1 UNTIL I > N.
TAKE-ONE-TERM.
 DIVIDE 2 INTO TERMS(I) GIVING HALF-TERM REMAINDER RMDR.
 IF RMDR IS EQUAL TO 1: ADD TERMS(I) TO SUM.
```

Toujours le même programme, ci-dessous en langage Pascal. Ce langage, conçu comme un successeur au langage Algol, a été développé par Nicolas Wirth vers la fin des années 1960. Une des idées directrices durant le devéloppement de ce langage était de vouloir construire un langage dont la correction de toute expression pourrait être formellement établie. Ce langage était un des chevaux de bataille de la programmation structurée. Entre temps, il est utilisé principalement pour l'apprentissage scolaire de la programmation.

Voici alors le programme en Pascal :

```
program SumOddNumbers;

type Termindex = 1..100;
    TermArray = array[Termindex] of integer;
var myTerms: TermArray;

function SumOdds (n: Termindex; terms: TermArray): integer;
```

```
var i: Termindex;
   sum: integer;

begin
   sum := 0;
   for i:= 1 to n do
      if Odd(terms[i]) then
            sum := sum + terms[i];
   SumOdds = sum;
end;

begin
   myTerms[1]:23; myTerms[2] := 34; myTerms[3] := 7; myTerms[4] := 9;
   WriteLn (SumOdds (4, myTerms));
end.
```

Le langage Forth est un autre langage utile à écrire ce programme. Forth est un langage très compact, initialement développé (par Charles H. Moore en 1970) pour les astronomes du Smithonian Astrophysical Observatory, pour calculer des ephemerides, des orbites, des positions de satellites, etc. Ce langage est relativement près de la machine et est souvent utilisé pour des applications de contrôle de processus sur des micro-ordinateurs. Son nom est dérivé de “fourth”, anglais pour quatrième, comme dans “langage de quatrième génération”.

Ci-dessous une version en ce langage :

```
:SUMODDS
   0 SWAP 0
   DO
      SWAP DUP 2 MOD
            IF +
                  ELSE   DROP
                  THEN
   LOOP
:
```

Le calcul en Forth est, comme celui des calculateurs HP, complètement orienté autour d’une pile unique. Voici une trace de l’activation de

23 34 7 9 4 SUMODDS

instruction	pile	commentaires
23	23	
34	23 34	
7	23 34 7	
9	23 34 7 9	

4	23	34	7	9	4			
SUMODDS	23	34	7	9	4			appel de SUMODDS
0	23	34	7	9	4	0		
SWAP	23	34	7	9	0	4		
0	23	34	7	9	0	4	0	
DO	23	34	7	9	0			depilage valeurs de contrôles de boucle
SWAP	23	34	7	0	9			
DUP	23	34	7	0	9	9		
2	23	34	7	0	9	9	2	
MOD	23	34	7	0	9	1		
IF	23	34	7	0	9			TopStack = 1 alors faire IF à THEN
+	23	34	7	9				
ELSE	23	34	7	9				aller après THEN
DROP	23	34	7	9				skipped
THEN	23	34	7	9				skipped
LOOP	23	34	7	9				retour vers DO
DO	23	34	7	9				
SWAP	23	34	9	7				
DUP	23	34	9	7	7			
2	23	34	9	7	7	2		
MOD	23	34	9	7	1			
IF	23	34	9	7				TopStack = 1: faire IF à ELSE
+	23	34	16					
ELSE	23	34	16					aller après THEN
DROP	23	34	16					skipped
THEN	23	34	16					skipped
LOOP	23	34	16					retour vers DO
DO	23	34	16					
SWAP	23	16	34					
DUP	23	16	34	34				
2	23	16	34	34	2			
MOD	23	16	34	0				
IF	23	16	34					TopStack = 1; faire ELSE à THEN
+	23	16	34					skipped
ELSE	23	16	34					
DROP	23	16						
THEN	23	16						
LOOP	23	16						retour vers DO
DO	23	16						
SWAP	16	23						
DUP	16	23	23					
2	16	23	23	2				
MOD	16	23	1					
IF	16	23						TopStack = 1; faire IF to ELSE
+	39							
ELSE	39							skipped
DROP	39							skipped
THEN	39							skipped

LOOP	39	plus d'itérations
:	39	retour de SUMODDS
	<pile vide>	imprime le résultat

Le langage APL est un langage particulièrement adapté au calcul numérique. Développé par Kenneth Iverson au laboratoire de recherche de IBM, au début des années 1960, c'est un exemple d'un langage conçu spécialement pour des applications particulières : des calculs mathématiques.

Voici alors une version bien compacte du même programme exemple, cette fois en APL :

```
    ∇ SUM ← SUMODDS TERMS
[1] SUM ← +/(2 | TERMS) / TERMS
    ∇
```

et le lancement :

```
SUMODDS 23 34 7 9
```

a comme effet :

TERMS	←	23	34	7	9	affectation des valeurs initiales
(2 | TERMS)	←	1	0	1	1	tableau des modulos
(2 | TERMS)/TERMS	←	23		7	9	compression des deux tableaux
+/(2 | TERMS)/TERMS	←	23	+	7	+ 9	réduction par addition
SUM	←	39				affectation du résultat

Et encore ce même programme, maintenant écrit en langage LISP. Avec Fortran, Lisp (acronyme pour LISt Processing language) est le plus ancien des langages de programmation toujours utilisé. Il a été développé en 1959 par John McCarthy au MIT. C'est le langage le plus utilisé en intelligence artificielle. C'est également le langage qui connaît le plus de dialectes différents, ceci est dû à la simplicité d'en construire un interprète et un compilateur. Il y a même des machines dont le langage machine est un dialecte de LISP ! Un programme en LISP pure est un ensemble de définition de fonctions et d'appels de ces fonctions.

Voici alors notre programme écrit en LISP :

```
(DE SUMODDS (TERMS)
 (COND
   ((NULL TERMS) 0)
   ((ODD? (CAR TERMS)) (+ (CAR TERMS)(SUMODDS (CDR TERMS))))
   (T (SUMODDS (CDR TERMS)))))
```

et le lancement :

```
(SUMODDS '(23 34 7 9))
```

produira une exécution similaire à la trace que voici :

```
            (+ 23 (SUMODDS '(34 7 9)))
                (+ 23 (SUMODDS '(7 9)))
                    (+ 23 (+ 7 (SUMODDS '(9))))
                        (+ 23 (+ 7 (+ 9 (SUMODDS '()))))
                    (+ 23 (+ 7 (+ 9 0)))
                (+ 23 (+ 7 9))
            (+ 23 16)
= 39
```

Voici, une dernière version de ce programme exemple : cette fois écrit en langage Smalltalk. Smalltalk est un langage issu de Simula, SketchPad, Plasma, Logo et Flex. Sa première version a été développé par Alan C. Kay, entre 1970 et 1972, au Xerox Palo Alto Research Center. La programmation Smalltalk se fait par la définition de structures de données organisées hiérarchiquement et l'ajout à ces structures de données (qui s'appellent des classes) de fonctionnalités actives. Le programme sera alors un ensemble de telles classes et l'envoie de messages à des instances de classes. Les maître mots de la programmation Smalltalk sont : instanciation, envoie de messages, héritage.

Dans ce langage, la fonctionnalité décrite ci-dessous sera ajoutée aux capacités de la classe Collection. On dira, que le code ci-dessous est une méthode (de la classe Collection) qui décrit comment une instance de cette classe doit répondre si on lui demande sa sumOfOddNumbers.

```
sumOfOddNumbers
        "d'abord un exemple d'appel :"
        "#(23 34 7 9) sumOfOddNumbers"
        ^  self inject: 0 into: [:x :y | x + (y odd
                    ifTrue: [y]
                    ifFalse: [0])]
```

Finalement, un exemple de programme dans un langage qui n'a pas de commandes mais qui consiste entièrement en des déclarations. Le langage-type pour cette sorte de langage non-procédural est le langage Prolog. Prolog a été développé par Alain Colmerauer au début des années 1970 à l'Université Marseille Luminy. En Prolog, un programme ne donne pas des instructions explicites disant comment exécuter une instruction; il exprime juste des relations et fait des inférences basées sur ces relations. Par exemple une relation comme :

produit (hauteur, largeur, surface) →...

décrit l'égalité surface = hauteur * largeur, mais elle ne spécifie pas que la hauteur et la largeur sont les quantités données initialement, ou que la quantité demandée est la surface. En effet, cette relation peut servir, étant données deux quantités, à trouver celle qui manque (soit la hauteur, la largeur ou la surface). Par exemple, si l'on demande ensuite :

produit(20, x, 200);

on obtiendra le résultat x=10 et si l'on demande :

produit(20,10,x);

le résultat sera, bien naturellement, x = 200.

Voici un mini programme Prolog permettant d'interroger une base de données :

```
les déclarations de base
homme(Pierre) → ;
homme(Henry) → ;
femme(Vanessa) → ;
femme(Deborah) → ;
femme(Delphine) → ;

pere(Pierre, Henry) → ;
pere(Henry, Vanessa) → ;
mere(Deborah, Vanessa) → ;
mere(Delphine, Henry) → ;
etc...
ensuite les relations générales :
parent(x, y) →
   pere(x,y);

parent(x, y)→
   mere(x, y);

enfant(x, y)→
   parent(y, x)

fils(x, y)→
   enfant(x, y)
   homme(x);

fille(x, y)→
   enfant(x, y)
   femme(x);
```

```
frere-ou-soeur(x, y)→
    pere(p, x)
    pere(p, y)
    mere(m, x)
    mere(m, y)
    dif(x, y);

frere(x, y)→
    frere-ou-soeur(x,y)
    homme(x);
etc. ...
```

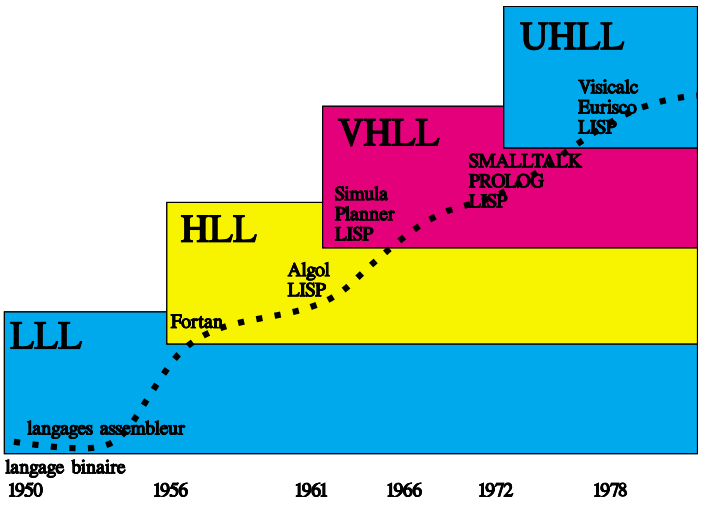
ensuite, on peut demander des questions du style “qui sont les parents de Vanessa ?”, ce qui se dit on Prolog :

```
parent(x, Vanessa);
    et la machine répond alors :
    {x = Henry}
    {x = Deborah}
```

ou encore “qui sont les parents de qui ?” qui se dit en Prolog :

```
parents(x, y);
    et la machine répond alors :
    {x = Henry, y = Vanessa}
    {x = Deborah, y = Vanessa}
    {x = Delphine, y = Henry}
    {x = Pierre, y = Henry}
```

Voici une vue condensée sur le développement des langages de programmationn :



1946	Konrad Zuse	Plankalkül
1949		Short Code
1951	Grace M. Hopper	A-0
1957	John Backus	Fortran
1958	John McCarthy	LISP 1.0
	John Backus	Fortran 2
1959	John McCarthy	LISP 1.5
	CODASYL	définition de Cobol
1960		Algol-60
	Kenneth Iverson	APL
1962		Fortan IV
	Ralph E. Grieswold & Ivan P. Polonsky	Snobol
1964	John G. Kemeny & Thomas E. Kurtz	Basic
		PL/1
1966		Fortran 66
		LISP 2
	Wallace Feuerzeug & Seymour Papert	LOGO
1967		Snobol 3
1968		Algol 68
	Alan Kay	Flex
1969	Niklaus Wirth	Pascal
1970	Charles Moore	Forth
1972	Alain Colmerauer	Prolog
	Alan Kay	Smalltalk-72
	Dennis Ritchie	C
1975	Guy L. Steele & Garry J. Sussman	Scheme
	Carl Hewitt	Plasma
1977	Niklaus Wirth	Modula
1978	Aho, Weinberger & Kernighan	AWK
1980	Adele Goldberg & Co	Smalltalk-80
	Bjarne Stroustrup	C with Classes
1982		Postscript
	Jean Ichbiah & Co	Ada
1983	Rick Mascitti	C++
1986		Smalltalk/V
	Bertrand Meyer	Eiffel
1988	Danny Bobrow & Co	CLOS
	Niklaus Wirth	Oberon
1990		Fortran 90
	Kenneth Iverson	J
1993	ANSI-X3J4.1	object oriented Cobol (!)