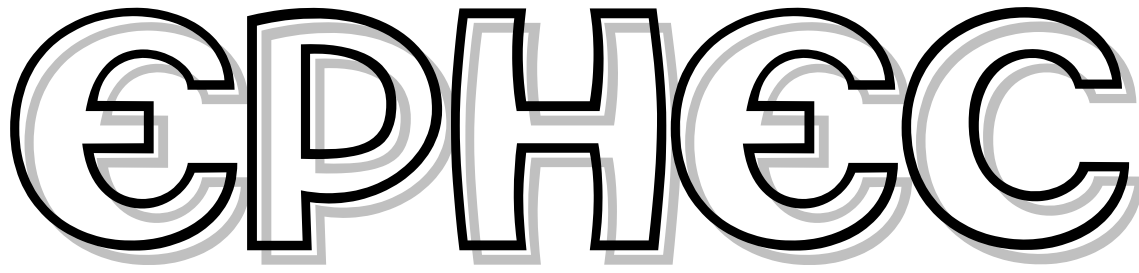


ÉCOLE PRATIQUE DES HAUTES ÉTUDES COMMERCIALES



INSTITUT D'ENSEIGNEMENT SUPÉRIEUR DE TYPE COURT

PRINCIPES DE PROGRAMMATION

ÉLÉMENTS D'ALGORITHMIQUE

PARTIE I : BASES DU RAISONNEMENT

TABLE DES MATIÈRES

1. AVANT-PROPOS.....	5
2. INTRODUCTION	6
3. LE CONCEPT D'AUTOMATE.....	9
4. LE LOGICIEL ROBOTPROG.....	11
4.1. PRÉSENTATION	11
4.2. L'ENVIRONNEMENT	11
• ... D'EXÉCUTION	11
• ... DE DÉFINITION	12
5. D'UN AUTOMATE FIGÉ	15
5.1. LES CONCEPTS DE BUT ET D'ACTION	15
5.2. LE CONCEPT DE SÉQUENCE.....	16
5.3. LE CONCEPT DE PROCÉDURE	17
6. ... VERS UN AUTOMATE ADAPTABLE	23
6.1. LES CONCEPTS D'ÉTAT ET DE CONTRÔLE	23
A) NOTION D'ÉTAT	23
B) NOTION DE CONTRÔLE	25
6.2. LE CONCEPT DE RÉPÉTITION	28
A) LA BOUCLE « TANT QUE ... »	30
B) EXERCICES.....	35
6.3. PAUSE ... BASES D'UN LANGAGE ALGORITHMIQUE COHÉRENT (1).....	37
A) L'INSTRUCTION CONDITIONNELLE (ALTERNATIVE) SI ... ALORS ... SINON	38
B) L'INSTRUCTION DE RÉPÉTITION TANT QUE	39
C) LE CONCEPT D'INSTRUCTION : CONCLUSION PROVISOIRE	39
D) L'ALGORITHME PRINCIPAL ET LES PROCÉDURES	40
E) LES COMMENTAIRES	41
6.4. EXEMPLES ET EXERCICES : D'AUTRES AUTOMATES ADAPTABLES	42
6.5. LES CONDITIONS MULTIPLES	53
A) LE PRINCIPE	53
B) ... UN EXEMPLE ROBOTPROG.....	54
C) ... UN AUTRE EXEMPLE SUR UN AUTRE AUTOMATE	56
D) EXERCICES.....	58
7. VALEURS, ÉTATS ET CONDITIONS	63
7.1. NOTION DE VALEUR.....	63
7.2. CONDITIONS SUR LES VALEURS : OPÉRATEURS DE COMPARAISON ET ÉTATS	64
7.3. NOTION DE TYPE DE DONNÉES	68

7.4. OUTILS DE MÉMORISATION	69
A) CONSTANTES.....	69
B) VARIABLES	69
C) AFFECTATION	70
7.5. EXERCICES	73
7.6. INTERACTION AVEC LE « MONDE EXTÉRIEUR »	75
A) AFFICHAGE DE MESSAGES	75
B) LECTURE DE DONNÉES	77
7.7. PAUSE ... BASES D'UN LANGAGE ALGORITHMIQUE COHÉRENT (2).....	79
A) LES TYPES, CONSTANTES ET VARIABLES.....	79
B) L'AFFECTATION.....	80
C) LES INSTRUCTIONS D'ENTRÉE/SORTIE	80
D) LA STRUCTURE D'UN ALGORITHME	81
7.8. EXERCICES	82
8. L'ORDINATEUR, AUTOMATE PROGRAMMABLE	83
8.1. ARCHITECTURE	83
A) COMPOSANTS.....	83
B) LANGAGE-MACHINE	84
C) AUTOMATE STATIQUE : FONCTIONNEMENT.....	85
D) AUTOMATE ... ADAPTABLE ?	88
8.2. LANGAGES DE PROGRAMMATION	93
A) DU LANGAGE-MACHINE AU LANGAGE D'ASSEMBLAGE	93
B) DU LANGAGE D'ASSEMBLAGE AUX LANGAGES DE PROGRAMMATION	95
9. DE L'ALGORITHME ABSTRAIT AU PROGRAMME CONCRET	97
9.1. PRINCIPE ET EXEMPLES	97
A) DÉMARCHE : DE L'ALGORITHME ABSTRAIT À L'ALGORITHME CONCRET	97
B) LANGAGE PASCAL (1972)	99
C) LANGAGE FORTRAN (1954)	102
D) LANGAGE COBOL (1959)	105
E) LANGAGE C (1973)	107
9.2. NOTION DE FONCTION	110
A) PRINCIPE	110
B) LANGAGE PASCAL	112
C) LANGAGE C	113
D) LANGAGE PERL (1987).....	114
E) LANGAGE PYTHON (1991)	115
F) LANGAGE RUBY (1995)	116
9.3. EXERCICES	117
10. EXERCICES DE SYNTHÈSE	119

1. AVANT-PROPOS

« ... Il est admis maintenant que l'informatique est une science, avec son objet spécifique : l'information et ses traitements, avec ses méthodes, ses outils (l'ordinateur étant le plus important, mais l'informatique n'est pas plus la science de l'ordinateur que l'astronomie n'est celle du télescope...). En tant que telle, elle fait partie du patrimoine culturel de la fin du 20^{ème} siècle. Elle a sa place dans tout enseignement scientifique. Mais là encore, on n'atteint qu'une partie de la réalité. L'astronomie est une science, et son importance est grande si l'on éprouve la moindre curiosité quant à l'univers qui nous entoure. Cela ne suffit pas à lui valoir une place de choix au lycée.

C'est pourquoi le colloque de Sèvres soutenait l'introduction de l'informatique parce que « une de ses caractéristiques est de créer chez les élèves une mentalité algorithmique, opérationnelle, organisatrice, laquelle est souhaitable pour bien des disciplines ». Plus que l'objet de l'informatique (l'information), c'est sa méthode, son esprit, qui lui valent l'attention qu'on lui porte aujourd'hui. C'est ce que je voudrais essayer de dégager ici.

L'ACTIVITÉ ALGORITHMIQUE

La notion d'algorithme est loin d'être claire. Dans de nombreux ouvrages on en trouve une définition qui ne permet pas de la distinguer de la notion de programme. Elle lui est pourtant de beaucoup antérieure. Prenons ici le mot « algorithme » comme désignant l'ensemble des opérations qui, appliquées à des données acceptables, fourniront la solution cherchée.

On pense aussitôt à des exemples pris en mathématique. Ainsi, dès le plus jeune âge, on fait acquérir aux enfants la pratique de l'algorithme d'addition dans notre système de numération. Ils apprennent à ajouter des nombres d'un chiffre (en comptant sur les doigts, avec un boulier, ou mieux en sachant par cœur une table d'addition). Puis ils apprennent à additionner 2 nombres de 2 chiffres, en utilisant le mécanisme de la retenue. Puis ils apprennent à généraliser pour l'addition d'un nombre quelconque de chiffres. Je ne suis pas très sûr qu'à aucun moment de leur scolarité on leur explicite ce dernier algorithme. Il faut reconnaître qu'il est particulièrement désagréable à énoncer.

De la même façon, les élèves apprendront à trouver le milieu d'un segment en traçant deux arcs de cercle et leur corde commune. Ont-ils conscience que c'est aussi un algorithme ? Éloignons nous des mathématiques. La conjugaison d'un verbe de la langue française relève d'un algorithme implicite. On donne le modèle de conjugaison de verbes types. Pour conjuguer un verbe quelconque, si ce n'est pas un verbe type, il faut découvrir la transformation qui fait passer de l'infinitif à la forme désirée, et l'appliquer à ce verbe.

Pas plus que pour l'addition, je ne crois pas que l'on explicite jamais un tel algorithme. La difficulté réside en fait dans l'absence de langage pour le dire. Un algorithme, c'est nécessairement un ensemble de règles de manipulation formelle correspondant à un niveau d'abstraction pas toujours facile à exprimer ou à appréhender pour de jeunes esprits.

L'informatique ne peut laisser ses algorithmes dans l'ombre. Les langages de programmation sont là pour les dire et le recours aux ordinateurs exige qu'on les dise.

On peut conserver l'attitude empirique consistant à les mettre en évidence sans les justifier. C'est une pratique beaucoup trop courante en informatique, et je ne peux que me réjouir des efforts entrepris par Dijkstra, Floyd, Wirth pour tenter de convaincre que l'on ne peut sérieusement continuer dans cette voie. « Essayer un programme sur des données particulières peut servir à prouver qu'il contient des erreurs, jamais qu'il est correct... »

Il faut bien voir où réside la difficulté. J'ai dit que l'on ne possède pas de bon langage pour dire les algorithmes. Faute de quoi, les langages de programmation ont joué un rôle de substitut. Mais s'ils sont aptes à dire les algorithmes, ils font peser sur eux les contraintes liées à la technologie des ordinateurs : nature séquentielle des instructions, place limitée en mémoire... Dans cette langue, on n'obtient pas la forme d'un algorithme la plus adaptée à sa démonstration...

Faut-il se décourager, et proclamer que rien ne presse, l'informatique en étant encore à chercher comment dire les algorithmes, et comment en justifier leur valeur. On s'en occupera au lycée plus tard. Je ne pense pas que l'on puisse soutenir raisonnablement ce point de vue. Il y a deux choses dans l'activité algorithmique : la mise en place de l'ensemble de règles efficaces et la preuve qu'elles sont efficaces. ...»¹

Jacques ARSAC

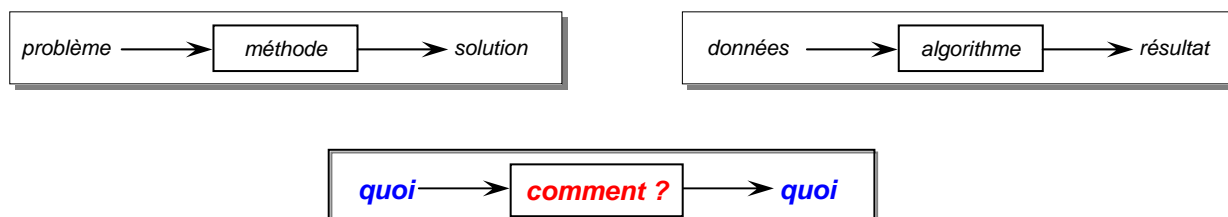
Directeur de l'Institut de Programmation de Paris

2. INTRODUCTION

J. Arzac le dit très bien dès 1976, la définition de ce qu'est un algorithme est relativement simple :

... ensemble des opérations qui, appliquées à des données, fourniront la solution cherchée ...

depuis lors, on en trouve bien d'autres, mais les meilleures d'entre elles expriment en fait toujours la même idée, celle du « *que faire ?* » et du « *comment faire ?* » mais en d'autres termes², p.ex. :



selon Courtin et Kowarski (en 1989)

« un algorithme est une suite d'actions que devra effectuer un automate pour arriver, en un temps fini, à un résultat déterminé à partir d'une situation donnée »

selon Aho et Ullman (en 1993)

« un algorithme est une spécification précise et non ambiguë d'une séquence d'étapes pouvant être exécutées d'une façon automatique. »

Mais le chemin est parsemé d'embûches : il faut que la méthode qui permet de passer du problème à sa solution soit 'bonne' :

selon Cormen, Leiserson, et Rivest. (Introduction à l'algorithmique. Dunod, 1994.)

« un bon algorithme est comme un couteau tranchant —il fait exactement ce que l'on attend de lui, avec un minimum d'efforts. L'emploi d'un mauvais algorithme pour résoudre un problème revient à essayer de couper un steak avec un tournevis : vous finirez sans doute par obtenir un résultat digeste, mais vous accomplirez beaucoup plus d'efforts que nécessaire, et le résultat aura peu de chances d'être esthétiquement satisfaisant. »

¹ extrait de « CLEFS POUR L'INTRODUCTION DE L'INFORMATIQUE DANS L'ENSEIGNEMENT » paru en 1976 et qui reste si actuel !

² références extraites du cours d'Algorithmique (2005) de Maria MALEK, enseignante à l'EISTI (École Internationale des Sciences du Traitement de l'Information) à Cergy (France)

L'objet d'un cours d'Algorithmique³ est donc l'étude du « **comment** (bien) **faire** ? », c.-à-d. de la méthode (de la manière, de la 'recette', ...) en privilégiant le raisonnement et donc la logique (avec bien souvent une solide dose de bon sens, si souvent oublié) ...

... et c'est loin d'être gagné d'avance : « *Ce n'est pas parce que vous savez faire quelque chose que vous savez le faire faire par un autre.* ».

Le premier pas à franchir en algorithmique est donc d'observer notre propre raisonnement à l'œuvre ...

Quant à la question du « **quoi faire** ? » - étudiée du point de vue des données de départ et des données à obtenir - elle sera bien entendu abordée dans ce cours (elle s'imposera d'elle-même en temps opportun, mais nous nous restreindrons à des données 'simples') ; pour ses aspects plus 'pointus', elle fera l'objet d'un autre cours spécifique 'Organisation et Structure des Données'⁴

Reste entière la question : comment 'dire ou écrire' les algorithmes, et en particulier comment enseigner l'algorithmique ? ... toujours selon J. Arsac :

«La difficulté réside en fait dans l'absence de langage pour le dire. Un algorithme, c'est nécessairement un ensemble de règles de manipulation formelle correspondant à un niveau d'abstraction pas toujours facile à exprimer ou à appréhender ... »

Mais dira-t-on : « il y a les langages de programmation, pourquoi ne pas les utiliser ? »

Si écrire un programme pour le faire exécuter par l'ordinateur est bien - pour le programmeur - l'objectif final, cette démarche doit occuper la place qui lui est réservée : la dernière !

Un programme n'est que la réalisation (l'implémentation, dira-t-on) d'un algorithme au moyen d'un langage particulier ...

... et puis quel langage choisir ? Fortran ?, Cobol ?, Pascal ?, C ?, C++ ?, Java ?, C# ?, Perl ?, Python ?, Ruby ?, PHP ? , ... ?????? ☺.

Durant les cinquante premières années de l'informatique moderne (1950 - 2000), ce sont plus de 2500 langages de programmation différents qui ont vu le jour !⁵

Voilà pourquoi nous choisirons ici d'utiliser ce que nous connaissons le mieux (?) : le langage naturel, reprenant à notre compte ce conseil des pères fondateurs de l'algorithmique et de la programmation :

Réfléchissez d'abord, vous programmerez plus tard !

³ le mot vient du nom du mathématicien arabe Al Khuwarizmi (latinisé au Moyen Âge en Algoritmi), qui, au IX^e siècle écrivit le premier ouvrage systématique sur la solution de certaines équations mathématiques

⁴ ... au second semestre

⁵ rendez-vous au chapitre 8, page 81 ...

3. LE CONCEPT D'AUTOMATE

« Un **automate** est un dispositif se comportant de manière automatique, c'est-à-dire sans intervention d'un humain. Ce comportement peut être figé, le système fera toujours la même chose, ou bien peut s'adapter à son environnement. » (<http://fr.wikipedia.org/wiki/automate>)

« Un automate ... est une machine imitant les mouvements, les fonctions ou les actes d'un corps animé. Des origines jusqu'à nos jours, la création des figures animées, d'une complexité de plus en plus grande à mesure que se développent les sciences et les techniques, paraît avoir été - que le but en fût magique, religieux, scientifique ou récréatif - un des besoins élémentaires de l'humanité. Dès l'origine, l'homme semble avoir cherché à reproduire l'apparence et le mouvement des êtres de son milieu vital. Animer le monde qui l'entoure pour s'en rendre maître va être une des premières recherches d'une humanité qui attribuait aux images et à la parole une force magique. »

(<http://www.universalis.fr/encyclopedie/B922881/automate.htm>)

Il faut donc remonter à l'antiquité grecque (1^{er} siècle après J.-C.) pour trouver la trace des premiers automates : on doit en effet à Héron d'Alexandrie la réalisation de machines animées et la publication du premier 'Traité des automates'. (http://fr.wikipedia.org/wiki/Héron_d'Alexandrie)

Ensuite, grâce aux progrès constants des sciences mécaniques et en particulier de l'horlogerie, toutes sortes de machines automatiques ont été réalisées, tant pour l'industrie (p.ex. fin du 18^{ème} siècle les métiers à tisser : http://fr.wikipedia.org/wiki/Métier_Jacquard) que pour le divertissement (sous forme de jouets, d'instruments de musique, comme l'orgue de barbarie, ou encore d'objets pour spectacles (http://fr.wikipedia.org/wiki/Automate_d'art) ... Des artisans (les plus célèbres au 18^{ème} étaient Vaucanson et Jaquet-Droz <http://www.paperblog.fr/1720019/les-automates-des-jaquet-droz-le-film/>) perpétuent d'ailleurs toujours aujourd'hui cette tradition. :

<http://www.francoisjunod.com/index.php?id=377> (à voir !)

Quel rapport avec l'informatique et l'algorithmique en particulier ? parce que durant tout ce temps, certains essayaient de réaliser des automates spécifiques : les machines à calculer (depuis Blaise Pascal en 1642, <http://fr.wikipedia.org/wiki/Pascaline> en passant par Charles Babbage en 1823 http://fr.wikipedia.org/wiki/Charles_Babbage ...) jusqu'à ce que, depuis son apparition en 1948, l'ordinateur moderne se révèle le seul automate programmable universel inventé par l'homme ; quelle autre machine, en effet, peut quasi instantanément se transformer en machine à écrire, à calculer, en salle de cinéma, en salle de concert, en simulateur de vol, en laboratoire de photographie, en console de jeux, en encyclopédie consultable à la demande... ? « véritable 'super couteau-suisse', l'ordinateur peut simuler toutes les machines imaginables⁶ »

Cette capacité provient de presque quarante années de recherches (entre 1900 et 1937) par les plus grands savants en mathématiques ... pour enfin aboutir à l'idée émise par Alan Turing d'un automate abstrait disposant d'une unité centrale de traitement dotée d'une mémoire où données et raisonnement (algorithme) figureraient côte à côte et seraient représentés de la même manière (par des nombres).



J. von Neumann

Depuis l'ENIAC en 1948, tous les ordinateurs sont des 'machines de Turing' ... mais il serait injuste d'oublier celui qui permit de passer de la théorie à la réalisation concrète, John von Neumann, considéré comme le père (avec J. Eckert et John Mauchly) de l'architecture des ordinateurs modernes^{7 8}



A. Turing

⁶ dans ... « Origines des nombres et du calcul », n° spécial des Cahiers de Science et Vie, août-septembre 2009

⁷ Depuis 1966, le prix Turing (Turing Award en anglais) est annuellement décerné par l'Association for Computing Machinery à des personnes ayant apporté une contribution scientifique significative à la science de l'informatique. Cette récompense est souvent considérée comme l'équivalent du prix Nobel de l'informatique. (wikipedia)

⁸ L'Institute of Electrical and Electronics Engineers (le fameux IEEE, prononcez "aïe tripeli") décerne chaque année une médaille en l'honneur de von Neumann, la IEEE John von Neumann Medal

4. LE LOGICIEL ROBOTPROG

4.1. PRÉSENTATION

Destiné à l'apprentissage du raisonnement algorithmique (et très accessoirement de la programmation), RobotProg est un logiciel éducatif (dernière version disponible : version 1.1), réalisé par Corinne Quemme, gratuit et de durée illimitée, et s'exécutant aussi bien sur des plateformes Windows que MacOSX.

<http://www.physicsbox.com/indexrobotprogfr.html>

Il permet de manière simple et ludique de 'programmer' un petit robot et de découvrir les deux aspects fondamentaux de tout ordinateur moderne considéré comme un automate adaptable (aspects qui seront étendus à tout programme informatique) ...



- les actions élémentaires
- les états binaires

... et ceci à travers un 'langage' graphique proposant⁹ les outils de base de toute programmation structurée :

- les instructions d'action
- les séquences d'instructions
- les procédures
- les instructions de contrôle
- les instructions de répétition

... la question des données sera également abordée le moment venu, de leur nature et de leur mémorisation (ce qui permettra au robot p.ex. de se souvenir de sa position et/ou de son orientation de départ, ou de certaines étapes par lesquelles il passe ...)

4.2. L'ENVIRONNEMENT ...

- ... D'EXÉCUTION (nous commençons par la fin)

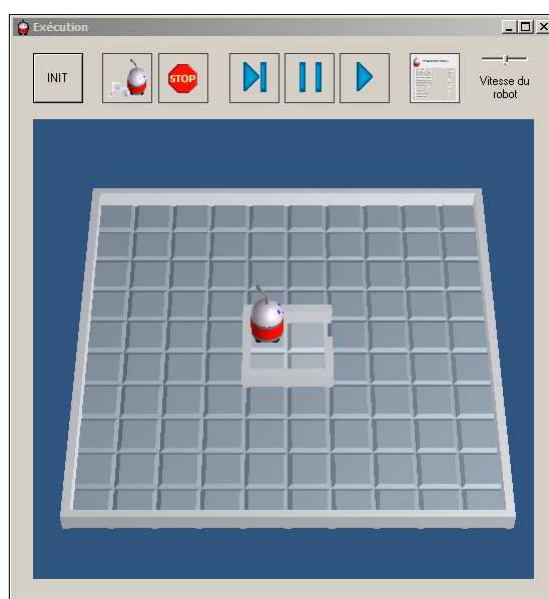
Par défaut, il ressemble à ceci :

La partie supérieure de la fenêtre propose divers boutons de commande pour - notamment - initialiser, lancer, suspendre, arrêter l'exécution du programme pilotant le robot ... on y reviendra

Le robot évolue sur un 'terrain' rectangulaire constitué de cases (le robot se déplace de case en case) et bordé d'un mur (infranchissable).

Par défaut, à chaque lancement du logiciel, le terrain est celui illustré ci-contre : le robot est dans une 'petite pièce' constituée de murs (avec une ouverture) et 'pavée' de manière spéciale.

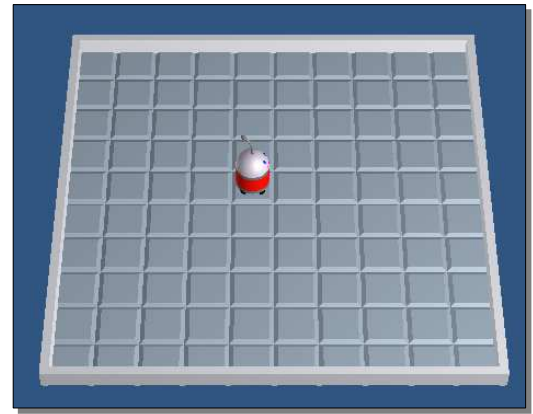
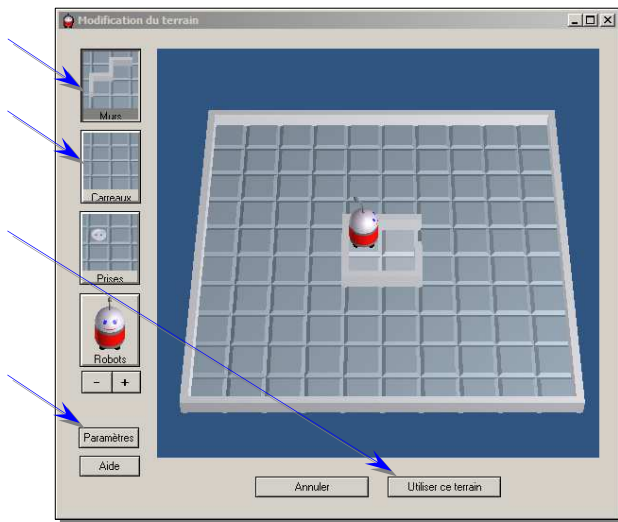
On reviendra (pour l'utiliser) sur cette configuration plus tard, nous allons - ce que permet le logiciel -, construire un terrain plus simple (sans la partie centrale) :



Menu > Terrain > Modifier

⁹ malheureusement il n'est pas parfait (avis personnel de l'auteur de ces lignes)

En utilisant les deux modes 'Murs' et 'Carreaux' et en cliquant sur les éléments du terrain, enlevons la pièce (ses murs et son pavage) ... pour obtenir ceci :



Cette configuration peut à ce moment être enregistrée sur disque (*Menu > Terrain > Enregistrer sous ...*) avec un nom au choix (et une extension .bog) ... enfin, un clic sur <Utiliser ce terrain> complète et termine le processus.

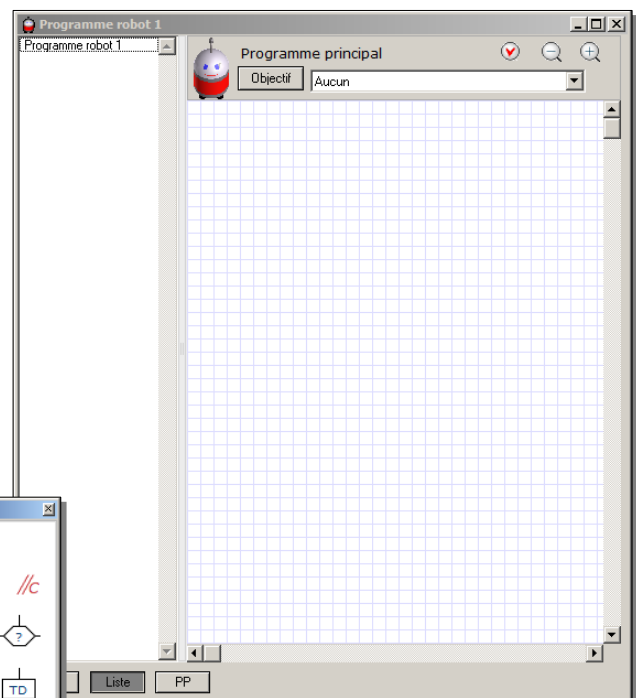
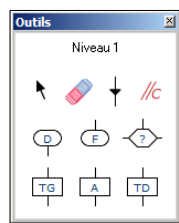
Comme il n'est pas possible de définir un terrain par défaut au lancement du logiciel (c'est celui illustré ci-dessus à gauche qui est imposé), il faudra passer par *Menu > Terrain > Ouvrir* pour utiliser un terrain enregistré. On peut aussi créer un nouveau terrain à partir de rien : *Menu > Terrain > Nouveau*

On remarquera que via le bouton <Paramètres>, il est possible de déterminer la taille du terrain (en nombre de cases dans les deux directions) ... et d'autres caractéristiques non essentielles à ce stade.

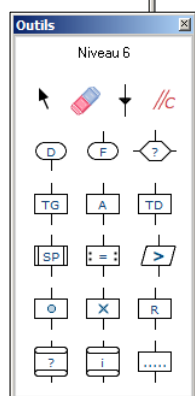
• ... de Définition

Pour 'faire faire' quelque chose au robot (pour définir la logique du comportement qu'il doit adopter), on dispose d'une double fenêtre :

- une palette d'outils (*ci-contre, détails plus loin dans le texte*)
- une fenêtre (*à droite*) pour la 'rédaction graphique' de l'algorithme ; à sa gauche, une liste qui comporte - par défaut et pour l'instant - le nom de cet algorithme (Programme robot 1) et à droite une grille de positionnement

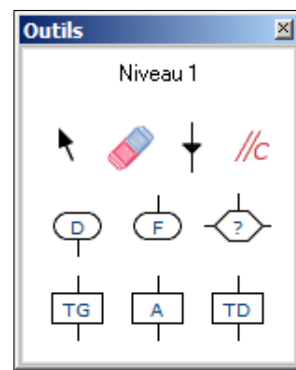


La conceptrice du logiciel a imaginé 6 niveaux d'apprentissage (d'élémentaire à compliqué, il faut croire ...), dès lors la palette va se garnir - en fonction du niveau choisi *Menu > Configuration > Niveau* - d'outils supplémentaires (sur des lignes supplémentaires).



Pour l'instant, restons au niveau 1 et explorons brièvement la palette correspondante :

- la première ligne comporte (de gauche à droite) un sélecteur, un effaceur, un traceur de ligne et de quoi écrire du texte (de commentaire/documentation)
- La deuxième ligne contient les outils de définition de <Début> (D) et de <Fin> (F) d'algorithme, (obligatoires parce que tout algorithme a un début et une fin !), et un outil de comparaison sur lequel on reviendra plus tard (concept d'état)
- La troisième ligne enfin propose les actions élémentaires dont est capable le robot (et avec lesquelles on va construire des actions plus complexes)¹⁰ : <Tourner à Gauche> (TG), <Avancer> (A), <Tourner à Droite> (TD)



¹⁰ dans la suite du texte, pour éviter les ambiguïtés, nous placerons le plus souvent les noms d'actions (et autres ...) entre <>

5. D'UN AUTOMATE FIGÉ ...

5.1. LES CONCEPTS DE BUT ET D'ACTION

Quand on dit 'automate figé' (on pourrait également utiliser le terme de 'statique'), nous ne voulons pas dire un automate (ici le robot) qui reste indéfiniment sur place, pas besoin d'algorithme ou de programme pour cela ! mais un automate qui va *'bêtement, sans se poser de question'* reproduire les instructions qu'on lui aura fournies.

Même l'automate le plus rudimentaire a un **but** (*ce qu'il y a à faire*), lequel peut être atteint à l'aide **d'actions élémentaires** (*avec quoi le faire*) et d'une **méthode**, un ordre précis dans lequel enchaîner ces actions élémentaires (*comment le faire*).

Pour parvenir au but assigné, la méthode va s'exprimer à l'aide d'instructions, en quelque sorte d'ordres ou de commandes d'exécuter telle ou telle action élémentaire.

Exemple : un lapin-jouet mécanique a pour but de jouer du tambour tout en avançant ; il dispose pour cela de mouvements élémentaires (pour les pattes 'supérieures' : lever la patte gauche, baisser la patte gauche, lever la patte droite, baisser la patte droite; pour les pattes 'inférieures' : avancer la patte gauche, avancer la patte droite) ; ces mouvements s'enchaînent dans le bon ordre grâce à un mécanisme (roues dentées, pignons, etc. ...) qui constitue les instructions d'un "programme" mémorisé. Notons en passant, que tous les automates ont également besoin d'une source d'énergie mécanique (p.ex. ressort) ou électrique (p.ex. pile ou secteur).



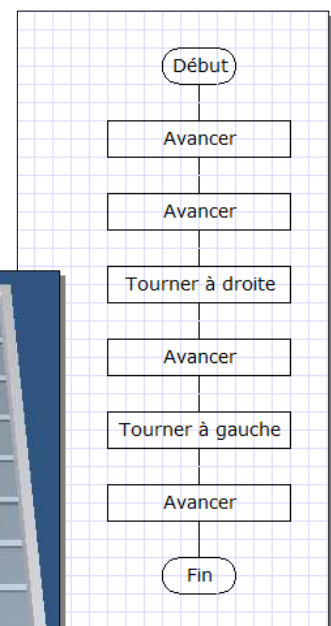
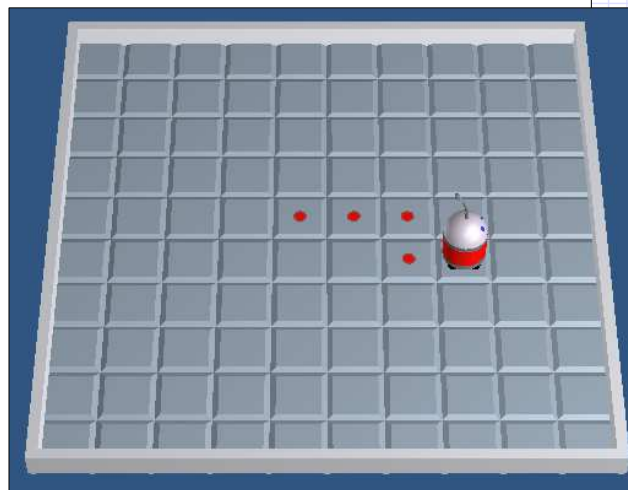
L'automate de RobotProg est capable – au niveau 1 - de trois actions élémentaires (Avancer, Tourner à Gauche, Tourner à Droite) ; en les enchaînant dans un ordre précis, nous lui donnons un comportement (il va démarrer, effectuer un certain parcours et s'arrêter); effectuer ce parcours est le but que nous lui assignons.

(Soyons plus précis : le robot ne peut se déplacer que dans la direction vers laquelle 'il regarde' : l'action <Avancer> le fait donc passer dans la case suivante selon cette direction ; les actions <Tourner à gauche> et <Tourner à droite> ont lieu sur la case en cours (pas de déplacement de case, donc), la gauche et la droite étant comptées par rapport à la direction courante du robot)

- Commençons par sélectionner les outils dans la palette et les déposer dans la fenêtre-programme : on commence obligatoirement par un début (démarrage), puis on enchaîne des actions les unes après les autres (ce que l'on appellera une séquence) jusqu'à la fin (arrêt).

- Après vérification (coche rouge au-dessus de la fenêtre-

programme), on passe à la fenêtre Exécution : cliquer sur <INIT> puis sur l'icône-robot ; sur la figure ci-contre, chaque point rouge indique le trajet parcouru, depuis la case de départ jusqu'à la case d'arrivée.



Il faut bien l'avouer, nous n'avons pas tout fait joué le jeu : au lieu d'assigner d'abord un but au robot et de chercher le moyen d'y parvenir, nous avons enchaîné quelques actions, 'pour voir' (et avons dit ensuite que c'était cela le but à atteindre ☺)

Il s'agissait ici simplement de 'faire connaissance'.

À l'avenir, nous respecterons à la lettre l'esprit de la démarche : **but** (*que faire*) – **méthode** (*comment faire*) – **actions** (*avec quoi faire*).

5.2. LE CONCEPT DE SÉQUENCE

Revenons un instant sur cette notion d'action : c'est ce que l'automate est capable de faire (ce sont ses potentialités), encore faut-il le lui faire faire ! On utilise pour cela des instructions (dans l'exemple de la page précédente, il y a ainsi six instructions utilisant les trois actions élémentaires dont est capable le robot).

Une instruction est l'outil permettant de spécifier l'action à exécuter

Reprenons notre exemple, on aurait dû commencer en précisant le but (la mission à réaliser) :

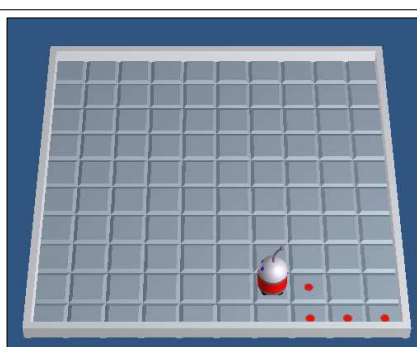
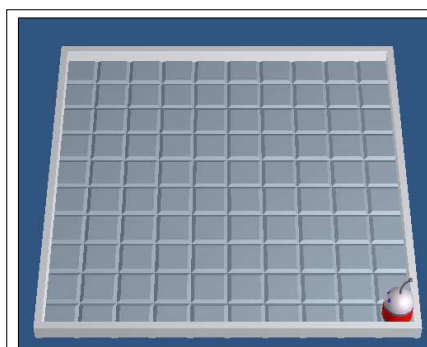
« amener de la manière la plus courte le robot sur une case située trois cases plus avant et une case plus à droite de sa position de départ, en conservant à l'arrivée sa direction initiale ... »

On constate alors qu'il y a plusieurs moyens équivalents (6 instructions, 3 actions) d'y parvenir, p.ex. : A A A TD A TG ou TD A TG A A A, (ou encore celle choisie ci-dessus : A A TD A TG A), ou d'autres encore¹¹ Mais attention, ce n'est pas parce qu'il y a chaque fois en tout quatre A, un TD et un TG qu'on peut les mettre dans n'importe quel ordre (essayez, A A A A TD TG, pour voir...).

C'est ici que la notion de séquence intervient :

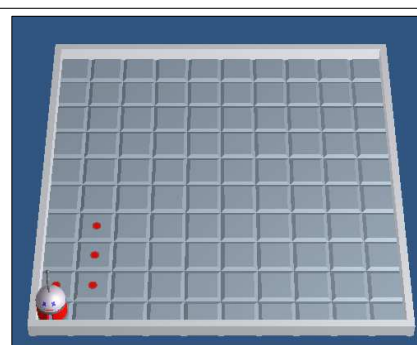
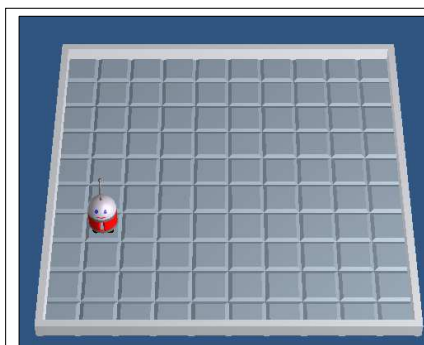
Une séquence est une suite ordonnée d'instructions qui seront exécutées dans l'ordre strict où elles ont été rédigées.

Il faut remarquer (on utilisera énormément ceci tout au long de notre apprentissage) qu'au



moment de l'initialisation <INIT>, l'utilisateur peut placer le robot n'importe où sur le terrain et lui donner n'importe laquelle des quatre orientations possibles (par clics successifs sur la case de départ souhaitée) ce qui permettra de tester le

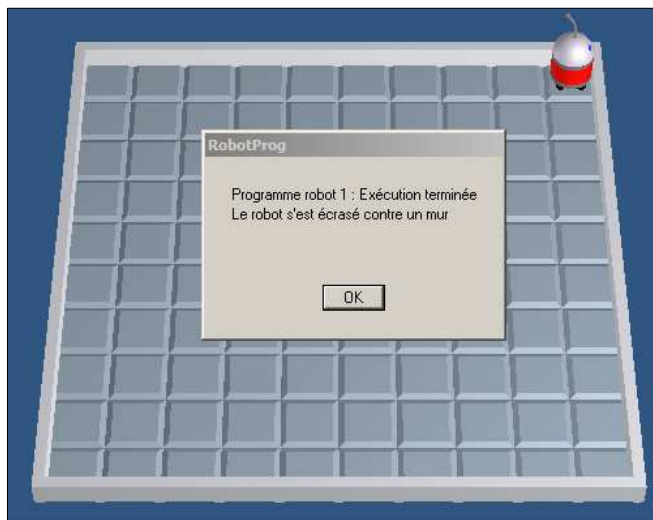
programme dans toutes sortes de configurations et de découvrir ainsi bien des surprises !



¹¹ Le lecteur est prié d'en dresser la liste complète comme exercice

On sent bien ce qui va arriver, à présent : que se passe-t-il si l'on exécute notre séquence (A A TD A TG A) si la case et l'orientation de départ est celle-ci (ci-contre) ?

Notre robot¹² est bête : il est juste capable d'exécuter 'à la lettre' les instructions données, on connaît de tels automates stupides : le distributeur de café qui fait couler le café alors qu'il n'y a plus de gobelets, qui a accepté votre pièce de 2 euros alors qu'il n'a pas de monnaie, certains humains se comportent comme tels pour peu qu'ils soient derrière un guichet, etc.¹³ ...



Il est donc temps de poser un **but premier** au robot (et forcément à celui qui décide du *comment*) : **éviter la destruction à tout prix !**

En attendant d'étudier la méthode pour y parvenir, il faut découvrir d'autres outils associés aux notions d'instruction et d'action : la séquence et la procédure.

5.3. LE CONCEPT DE PROCÉDURE

Avec l'action et la séquence, la notion de procédure est l'une des plus importantes pour structurer le raisonnement; pour l'illustrer, nous allons fixer un nouveau but à notre robot :

« Faire un aller et retour en ligne droite sur trois cases, de manière à se retrouver à la fin sur la même case qu'au départ et avec la même orientation. »

Facile ?

A
A
A
TD
TD
A
A
A
TD
TD

Bof ...

... il est temps de commencer à réfléchir de manière plus structurée, en prenant les choses 'par le haut' et non 'par le bas' ; et si nous reformulons d'abord et autrement le problème posé ?

avancer de 3 cases
faire demi-tour
avancer de 3 cases
faire demi-tour

C'est déjà mieux ..., plus explicite (quant au but fixé), plus lisible en tout cas ...

... bien entendu, ce serait sans doute encore mieux d'écrire

Faire 2 fois :
| avancer de 3 cases
| faire demi-tour

... mais oublions provisoirement cette dernière manière d'exprimer le problème (on y reviendra amplement) ... et regardons le contenu : « avancer de trois cases » et « faire demi-tour » apparaissent désormais comme deux "super-actions" indépendantes l'une de l'autre (faire demi-tour n'a rien à voir avec avancer), mais qui associées dans une séquence, permettent d'atteindre le but fixé.

¹² à moins que ce soit le programmeur ! un résultat désastreux renvoie souvent à celui-ci un sentiment d'incompétence, d'où l'intérêt de l'enseignement de l'algorithmique, des techniques à mettre en œuvre afin de 'penser mieux'

¹³ exercice : à partir de votre expérience, trouvez au moins deux exemples d'automates figés présentant de tels 'ratés comportementaux'

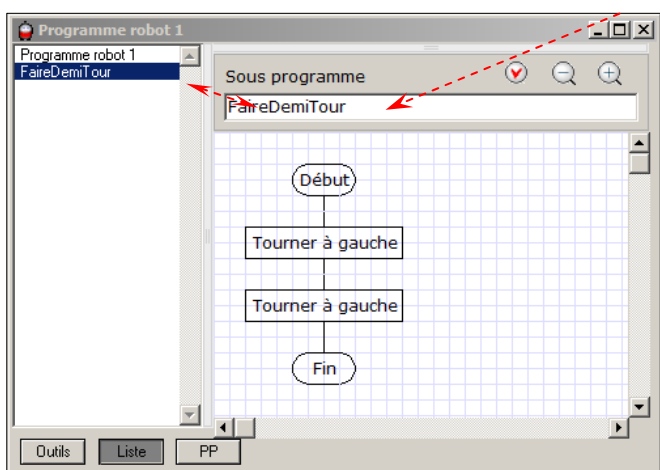
Commençons par 'apprendre' à notre robot comment faire demi-tour (on peut imaginer que c'est une situation qu'il pourra souvent rencontrer) : pour rappel, faire demi-tour consiste à se retrouver dans la direction opposée à la direction d'origine, sur la même case, sans effectuer de déplacement.

Compte tenu des seules actions disponibles (A, TG, TD), faire demi-tour (de la manière la plus simple et la plus rapide - et sans risque !) ne pourra s'écrire que de deux manières, soit TG TG soit TD TD (totalement équivalentes quant au résultat ... nous choisissons TG TG).

Plutôt que d'écrire explicitement ces deux instructions chaque fois qu'elles sont nécessaires, nous allons créer une nouvelle séquence et lui donner un nom : c'est une procédure.

Une procédure est une séquence d'instructions à laquelle on donne un nom et qui pourra être utilisée de la même manière que les actions élémentaires.

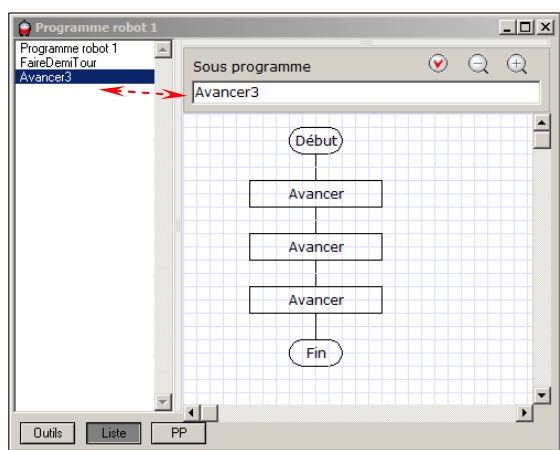
Pour ce faire, il faut (déjà !) demander le niveau 2, ce qui permet de créer cette procédure (RobotProg utilise à la place de procédure un nom équivalent, mais assez ancien : sous-programme) : Menu > Programmation > nouveau sous-programme



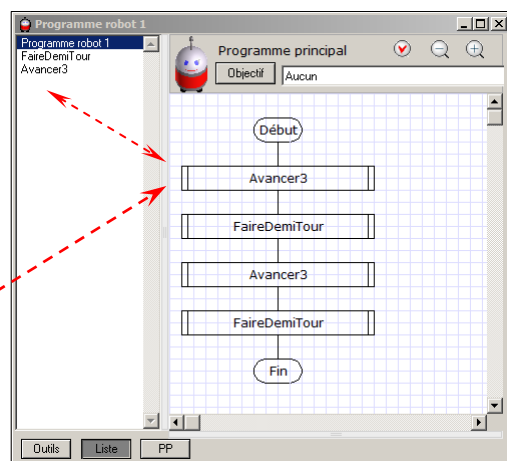
Il est essentiel de donner aux outils que l'on développe les noms les plus clairs et les plus explicites : en baptisant la procédure **FaireDemiTour**, on exprime ce qu'elle fait sans ambiguïté et en utilisant un verbe dans son nom, on précise son rôle d'action (comme Avancer et Tourner ...)

Remarque : les noms des nouvelles actions (procédures) ne peuvent pas contenir d'espace

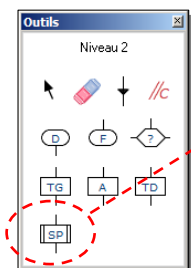
Dans la foulée, créons-en une autre, permettant au robot d'avancer en ligne droite de quelques cases (pour le moment, un nombre fixe, disons 3 cases, d'où le nom de la procédure **Avancer3**).



Il suffit maintenant d'écrire la séquence (algorithme) principale en utilisant – comme on le faisait avec les actions élémentaires – les deux sous-programmes (procédures) ;



pour ce faire, à partir du niveau 2, la palette d'outils propose l'icône SP (sous-programme) : il suffit de mettre le nom de la procédure que l'on veut exécuter



Il faut apprécier à sa juste mesure le progrès considérable que nous avons accompli ici : en écrivant une procédure, nous avons donné une nouvelle compétence à notre robot :

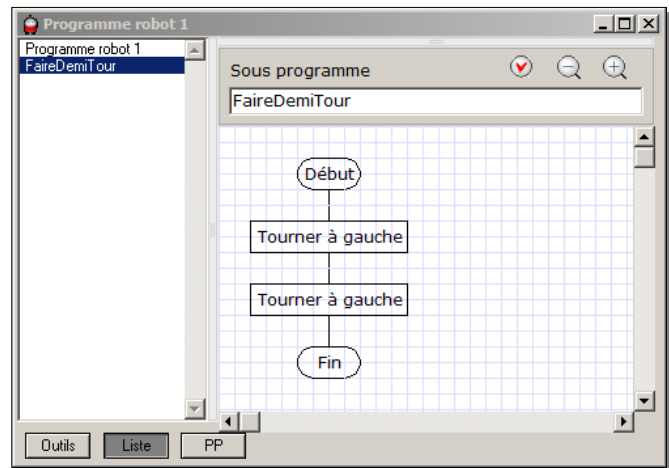
- une fois écrite, on va pouvoir l'utiliser (la procédure) en 'boîte noire' (c.-à-d. en retenant ce qu'elle fait, et en oubliant comment elle le fait) ...
- ... et on l'utilisera comme on utilise les actions élémentaires : elles aussi sont des procédures (on sait ce que fait <Avancer> sans pour autant en connaître les détails ... p.ex. démarrer un moteur, faire tourner des roues durant un certain temps ou sur une certaine distance ... arrêter le moteur)
- et surtout, cela permet d'exprimer la marche à suivre (la méthode, l'algorithme) en restant plus proche de la formulation du problème : on a gagné en abstraction !

On comprend dès lors mieux deux des termes qualifiant notre démarche de raisonnement : une approche impérative (sur base d'actions) et procédurale (possibilité de créer de nouvelles actions)

Autre aspect fondamental : l'indépendance :

on doit pouvoir à tout moment modifier une compétence (changer le contenu d'une procédure particulière) sans pour autant devoir intervenir dans l'algorithme principal;

il suffit par exemple de reprendre la procédure <FaireDemiTour> et de remplacer les TG par des TD : *la seule chose qui compte pour l'extérieur est que le comportement reste inchangé, la manière d'y parvenir n'a pas à être dévoilée, c'est le 'secret de fabrication' caché dans la procédure.*



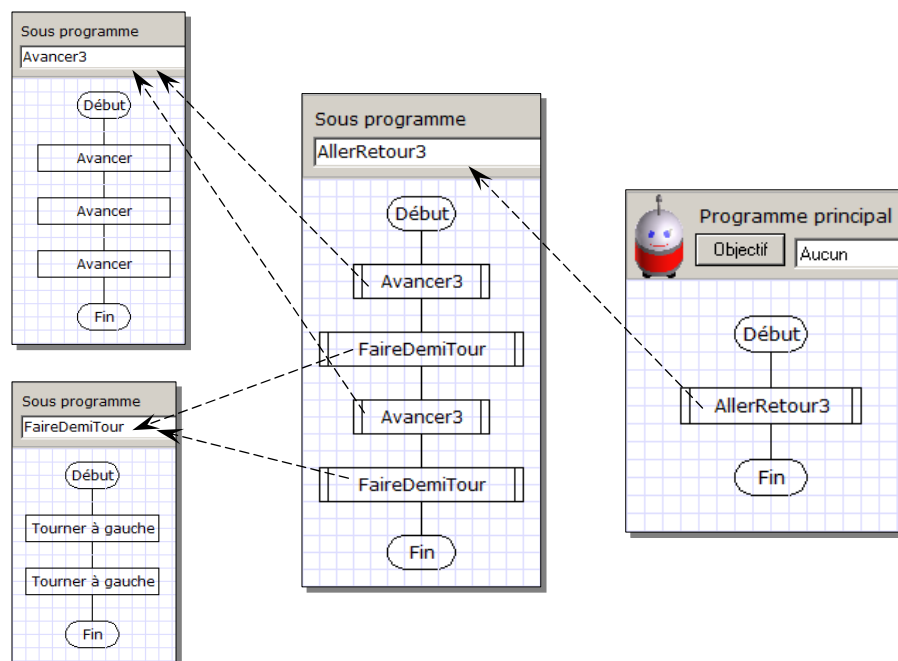
Nous disposons donc désormais des outils d'action ci-contre :

ce qui nous permet de rédiger une version finale structurée et procédurale de la mission

« Faire un aller et retour en ligne droite sur trois cases, de manière à se retrouver à la fin sur la même case qu'au départ et avec la même orientation. »

Actions
Avancer (A)
TournerÀGauche (TG)
TournerÀDroite (TD)
Procédure (SP)

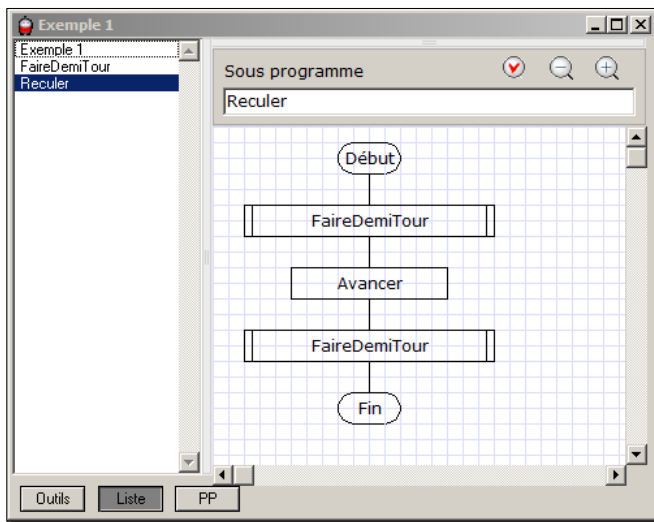
comme suit :



Bien entendu, dans une démarche de raisonnement structuré, les compétences acquises peuvent être réutilisées : une procédure doit donc pouvoir faire appel à une autre procédure.

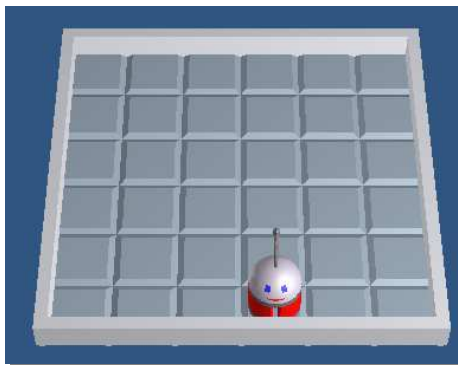
Fixons-nous un nouveau but :

« Faire reculer le robot d'une case »

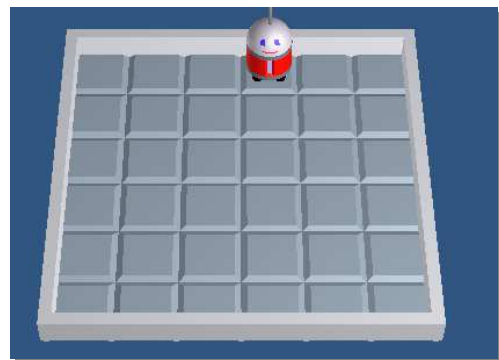


Comme il est vraisemblable qu'une telle action – qui ne fait pas partie des actions élémentaires du robot – sera sans doute utilisée à de multiples reprises, on en fait une nouvelle procédure **<Reculer>**, qui – on le voit clairement ci-contre – est une séquence faisant appel à des compétences connues qu'elles soient élémentaires (action <Avancer>) ou non (procédure <FaireDemiTour>)

Bien entendu, l'exécution de <Reculer> peut provoquer la même catastrophe que <Avancer> (mais cette fois si la position courante du robot est 'dos au mur') :..



cata si <Avancer>



cata si <Reculer>

il devient donc urgent de s'intéresser à la manière de transformer un automate figé en automate adaptable !

5.4. QUESTIONS & EXERCICES

- 1) Avec vos propres mots, donnez une définition/explication (en une phrase) des termes suivants :
 - instruction
 - automate
 - but
 - séquence
 - action
 - automate
 - procédure
 - pouvez-vous pour chacun en donner un synonyme ?
 - pouvez-vous les relier les uns aux autres à l'aide d'une phrase qui a du sens (donc met ces différents termes dans l'ordre) ?
- 2) En imaginant un robot ménager à qui on doit fournir un algorithme lui permettant de préparer du café ... si on dispose des actions élémentaires suivantes :
 - remplir la bouilloire au deux-tiers
 - débrancher la bouilloire
 - ouvrir le pot à café
 - mettre la bouilloire sous le robinet
 - fermer le tiroir
 - brancher la bouilloire sur le secteur
 - plonger la cuiller dans le pot
 - fermer le robinet
 - verser le contenu de la cuiller dans la tasse
 - attendre l'ébullition
 - verser une demi-tasse d'eau
 - ouvrir le tiroir
 - fermer le pot à café
 - prendre une cuiller à café
 - ouvrir le robinet
 - mettre ces différentes actions dans une¹⁴ bonne séquence pour que cet algorithme ait du sens et soit efficace (atteigne son but)
 - rédiger une version plus procédurale (structurée) de cet algorithme en regroupant ces étapes élémentaires dans trois procédures nommées

¹⁴ elle n'est pas nécessairement unique ...

6. ... VERS UN AUTOMATE ADAPTABLE ...

6.1. LES CONCEPTS D'ÉTAT ET DE CONTRÔLE

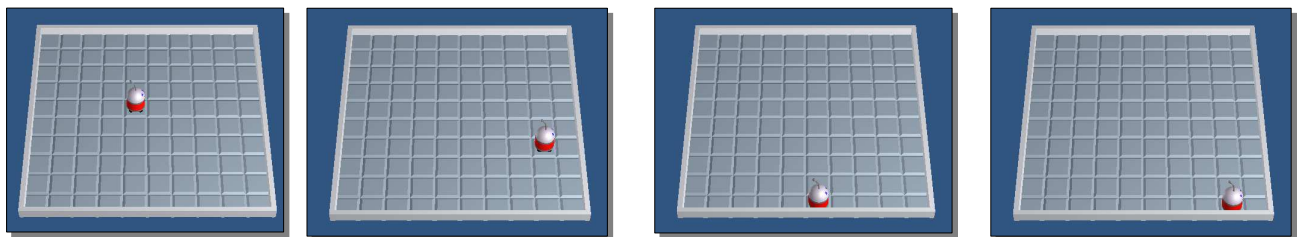
Nous disposons donc à présent de plusieurs outils : l'action élémentaire, la séquence d'actions (ordonnées) et la procédure (séquence nommée).

Pour aller plus loin (et ne serait-ce que pour atteindre notre but premier : permettre au robot de 'rester en vie à tout prix'), nous devons doter notre automate de possibilités de perception de son environnement (p.ex., comme l'action d'avancer se révèle catastrophique quand le robot est sur une case face à un mur, il¹⁵ doit pouvoir – avant d'exécuter l'action d'avancer – se 'poser la question' : *suis-je oui ou non face à un mur ? puis-je exécuter l'action sans risque ?*

a) Notion d'État

Avant de continuer, il faut remarquer que le robot est en interaction permanente avec son environnement :

- à tout moment il possède une position (une case sur la grille) et une orientation (sens de la marche) précises, (propriétés que nous désignerons par le terme d'état)
- et toute action entraîne toujours une modification de son (ou de ses) état(s) : <Avancer> modifie sa position (il change de case), <Tourner...> modifie son orientation (il change de direction) et <Avancer> aussi bien que <Tourner...> modifient son 'rapport au mur').



Ainsi, de gauche à droite : si sont illustrées la position et la direction actuelle du robot, quels sont les états qui sont modifiés par l'action d'<Avancer> d'une part et par l'action <Tourner à droite> d'autre part ?

	Situation 1		Situation 2		Situation 3		Situation 4	
Modification de l'état	A	TD	A	TD	A	TD	A	TD
position ?	<i>oui</i>	<i>non</i>	<i>oui</i>	<i>non</i>	<i>oui</i>	<i>non</i>	<i>oui</i>	<i>non</i>
orientation ?	<i>non</i>	<i>oui</i>	<i>non</i>	<i>oui</i>	<i>non</i>	<i>oui</i>	<i>non</i>	<i>oui</i>
contre mur ?	<i>non</i>	<i>non</i>	<i>oui</i>	<i>non</i>	<i>non</i>	<i>oui</i>	<i>oui</i>	<i>oui</i>

Pour permettre à un automate de s'adapter à son environnement, il faut :

- lui fournir de l'information quant à ses différents états (et à nouveau, peu importe la manière technique dont cela se réalise)
- lui donner la possibilité de tenir compte de ces états pour modifier éventuellement son comportement : il doit pouvoir prendre des décisions de manière autonome¹⁶

Proposons une définition (provisoire) du concept d'état :

Un état est une information relative au statut de l'automate vis-à-vis de son environnement

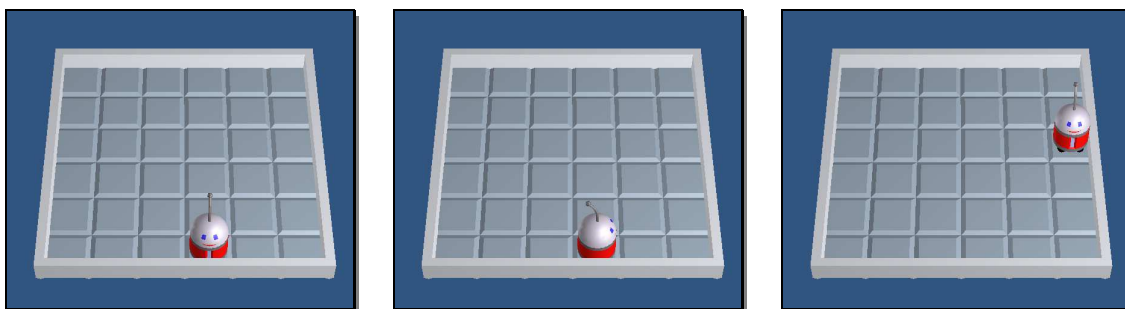
¹⁵ *il* doit « pouvoir » : amusant transfert de responsabilité : en fait, le rédacteur de l'algorithme doit offrir au robot de quoi lui permettre de « pouvoir »

¹⁶ à nouveau, cet usage du '*il*' : bien entendu, c'est ici aussi le rédacteur de l'algorithme qui doit lui fournir de quoi pouvoir prendre des décisions

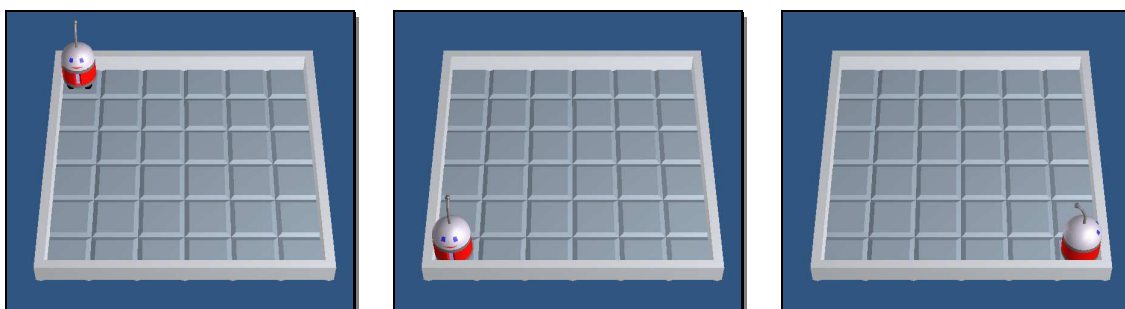
Nous allons utiliser à présent dans RobotProg certains de ces états, relatifs à la position du robot par rapport aux murs (les trois états ci-contre font partie des outils disponibles dès le niveau 1) :

- ces états sont particuliers : chacun correspond à une question sous la forme : « *est-ce que oui ou non ... ?* », question à laquelle il n'y a que deux réponses possibles « *oui, c'est vrai* » ou « *non, c'est faux* » : ce sont des états logiques¹⁷ (ou binaires) (qui n'offrent que deux possibilités)
- ainsi l'état <MurEnFace> est vrai lorsque le robot est sur une case et a 'le nez contre le mur'¹⁸; il est faux dans tous les autres cas ; il en va de même pour les états <MurAGauche> et <MurADroite> (par rapport à la direction courante du robot)

États
MurEnFace
MurAGauche
MurADroite



et bien entendu, comme dans le monde réel, le robot peut se trouver dans plusieurs états simultanément (au lecteur d'identifier lesquels dans les exemples ci-dessous ...)



remarque : si l'on jette un œil dans la documentation de RobotProg (Menu > Aide > Résumé du langage robot), on constate qu'il existe d'autres états (p.ex. <xRobot> et <yRobot> indiquant à tout moment l'état positionnel du robot sur son terrain-grille ...), mais patience ...

¹⁷ la (re)lecture du fascicule 'Éléments de Logique' dans le cours de Mathématique est vivement conseillée

¹⁸ insistons bien sur ce point : il est évident que compte tenu de son environnement rectangulaire fermé, le robot a toujours un mur 'en face' et 'à gauche' et 'à droite' ... les états <MurEnFace>, <MurAGauche> et <MurADroite> correspondent à des situations où le robot est 'contre' le mur et permettent de détecter "comment" il est contre le mur

b) Notion de Contrôle

Disposer d'états reflétant le statut d'un automate est bien, encore faut-il disposer d'outils pour en faire "quelque chose" ; c'est le rôle d'instructions particulières : les instructions de contrôle

Une instruction de contrôle est un outil permettant une prise de décision sur base d'un (ou plusieurs) états

L'instruction de contrôle la plus simple est appelée alternative¹⁹ : elle correspond à l'expression suivante dans le langage usuel :

- qu'entend-on par **condition** ? il suffit de prendre quelques exemples concrets

Si une **condition** est vraie,
Alors faire quelque chose
[**Sinon** faire autre chose]

si je réussis en première session je pars en vacances sinon j'étudie

s'il pleut je prends mon parapluie

s'il ne pleut pas j'irai à la plage sinon j'irai à la piscine

Une condition logique est la détermination du caractère vrai ou faux d'un état

bien entendu, le cerveau humain est très puissant et il utilise un langage très évolué (plutôt une langue qu'un langage, en fait) qui lui permet d'exprimer des idées et des raisonnements de manière souvent très compacte ... réécrivons-donc les exemples ci-dessus de la manière suivante

Si je réussis en première session Alors je pars en vacances Sinon j'étudie

Si il pleut Alors je prends mon parapluie

Si il ne pleut pas Alors j'irai à la plage Sinon j'irai à la piscine

- les termes 'si', 'alors' et 'sinon' servent à structurer la prise de décision : énoncer la **condition** (**Si**), décrire le comportement (**actions**) à adopter si cette condition est vraie (**Alors**) ou si elle est fausse (**Sinon**) ...

... on remarquera que les conditions (en rouge) sont bien des états logiques et que l'instruction de contrôle permet de spécifier le comportement (en vert) à adopter si l'état est vrai d'une part ou s'il est faux d'autre part ...

... et qu'on peut la 'retourner' grâce à la négation ; sont ainsi équivalentes :

Si je réussis en première session Alors je pars en vacances Sinon j'étudie

Si je ne réussis pas en première session Alors j'étudie Sinon je pars en vacances

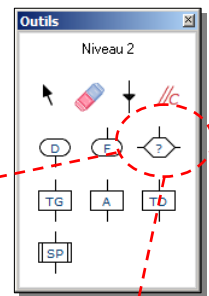
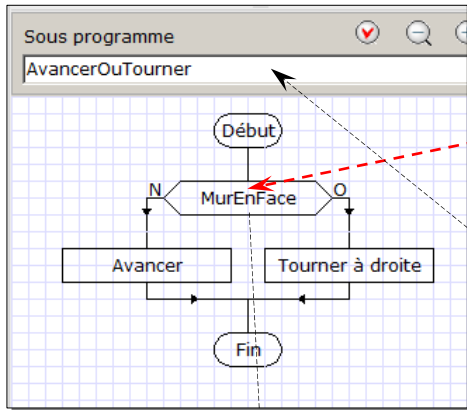
Poussons plus loin l'analogie entre le monde réel et le raisonnement algorithmique : quand on déclare « *jeudi, s'il ne pleut pas, j'irai à la plage sinon j'irai à la piscine* », on définit **maintenant** ce que l'on fera **dans le futur** en fonction de la météo ; le jour venu (jeudi) on évalue l'état de la météo : *pleut-il ?* et en fonction de la réponse oui/non, on adopte le comportement prévu.

Il en va de même quand nous rédigeons un raisonnement algorithmique : nous raisonnons **maintenant** pour le comportement **futur** de l'automate ; quand l'exécution aura lieu, l'automate devra se gérer seul, en toute autonomie, sur base de l'algorithme qu'on lui aura fourni (quand on envoie une sonde sur une planète lointaine, il n'y a pas un bus rempli de mécaniciens, électroniciens, informaticiens ... qui la suit ...)

¹⁹ on la désigne également par **instruction conditionnelle**

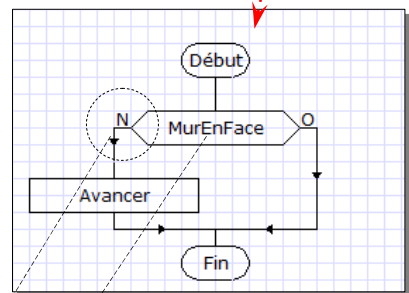
Une instruction de contrôle alternative définit quelle(s) instruction(s) seront exécutées sur base d'une condition logique

Dans RobotProg, l'alternative figure dans la palette d'outils sous forme d'un hexagone (représentation dépassée²⁰ héritée d'un passé lointain où les raisonnements se 'dessaient' à l'aide d'ordinogrammes) : il symbolise l'évaluation d'une condition (ici d'un des états disponibles) et dispose de deux 'pattes' horizontales permettant d'associer des séquences particulières aux valeurs vrai (Oui) et faux (Non) de cet état



Son utilisation est semblable à l'appel de procédure : on place l'outil sur la grille et on y encode le nom de l'état qui correspond à la condition.

En rédigeant (ci-dessus) une nouvelle procédure <AvancerOuTourner>, nous dotons notre robot d'une nouvelle compétence : se rendre compte par lui-même de son état par rapport au mur et d'éviter la destruction en changeant de direction (dans cet exemple du moins ...) ; on aurait pu imaginer une autre action ... par exemple ne rien faire, comme ci-contre dans la procédure <AvancerOuStopper> pour illustrer un Si ... Alors ... (sans la partie Sinon)



On observera dans les deux cas l'usage du principe de précaution : on évalue l'état avant de faire l'action qui comporte le risque de destruction (ici <Avancer>) : en effet, si on ne connaît pas la position et l'orientation du robot au départ, il faut prendre en considération le risque (jamais exclus) qu'il soit peut-être face au mur

AvancerOuTourner :
 Si **MurEnFace**
 Alors TournerADroite
 Sinon Avancer

AvancerOuStopper :
 Si non **MurEnFace**
 Alors Avancer

Remarque importante :

nous utilisons dans cette page deux moyens d'expression différents mais équivalents de l'alternative : l'ordinogramme (boîtes et flèches) et le langage naturel (si ... alors ... sinon); le lecteur est vivement encouragé à les examiner soigneusement ... afin de maîtriser le passage de l'un à l'autre ... et en particulier dans l'exemple de droite l'usage de la négation ne nous cachons pas ... notre but (à moyen terme, il y a encore des concepts essentiels à découvrir) est de privilégier l'usage du langage naturel pour l'expression des raisonnements

²⁰ avis qui n'engage que l'auteur de ces lignes, mais puisque nous sommes toujours dans une phase de début d'apprentissage et si cette manière 'visuelle' de représenter les choses peut aider à la compréhension ... soit !

c) Questions & Exercices

- 1) Avec vos propres mots, donnez une définition/explication (en une phrase) des termes suivants (et si possible, donnez-en un synonyme) :
 - condition
 - état
 - instruction de contrôle
 - alternative
- 2) Que pensez-vous de l'affirmation suivante
 - contrairement à une instruction d'action, une instruction de contrôle ne fait rien !
- 3) Reprenez le robot "machine à café" du chapitre précédent (§5.4 page 21) et considérez-le comme un automate (potentiellement) adaptable :
 - quels seraient selon vous les différents états dont il devrait disposer pour être entièrement automatique (donnez-leur des noms de manière adéquate) ?
 - en utilisant les instructions/actions élémentaires (fournies précédemment) et les états que vous venez de créer (au moyen de l'instruction de contrôle alternative Si ... Alors ... Sinon ...), réécrivez l'algorithme procédural permettant à ce robot de faire face à (toutes ?²¹) les situations qu'il pourrait rencontrer pour réaliser son but "faire du café"

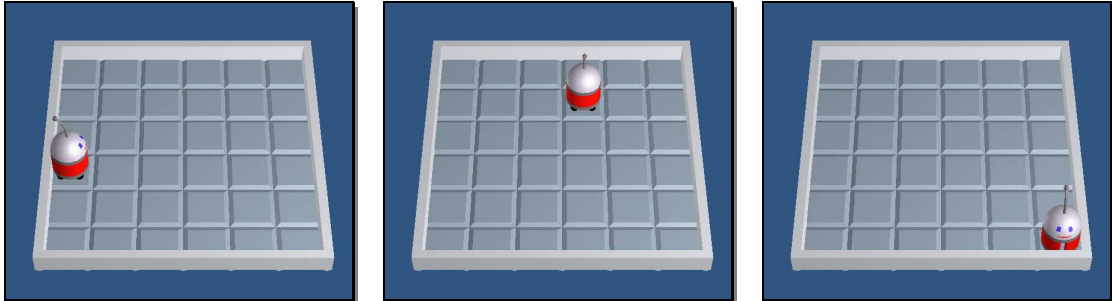
²¹ celles qui ont du sens ... par exemple il n'est pas nécessaire de commencer par "Si il n'y a pas de panne de courant Alors ...", votre automate ne fonctionnant qu'à l'électricité

6.2. LE CONCEPT DE RÉPÉTITION

Fixons-nous un nouveau but :

« *quelle que soit la position du robot, atteindre le mur d'en face et s'y arrêter* »

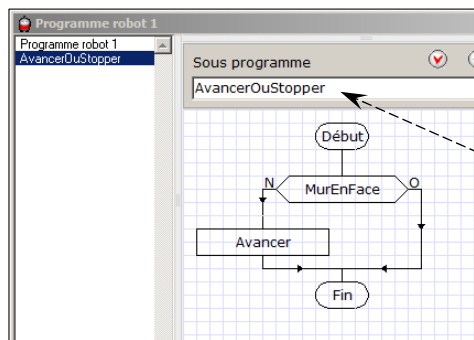
ci-dessous, quelques positions et directions de départ possibles (il y a 36 cases et 4 orientations, ce qui donne 144 possibilités de départ !) afin de bien poser le problème :



- la mauvaise approche consiste à regarder cela d'un point de vue humain, de l'extérieur, et dire : *à gauche, il suffit d'exécuter la séquence A A A A A, au centre d'exécuter la séquence A et à droite de ne rien faire !*

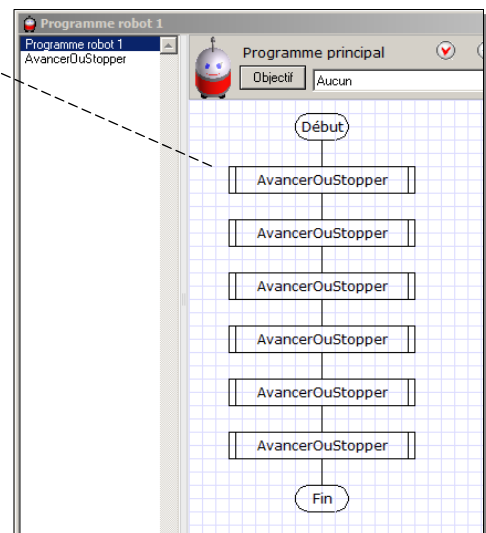
Il n'y a là aucun raisonnement, pas de réelle méthode, et surtout il manque la composante essentielle : la recherche de l'autonomie. On traite chaque situation-problème au cas par cas, ici et maintenant ; et, parce que l'on voit la grille et la position de départ, on se contente de faire (faire) le nombre de pas nécessaire.

- la bonne approche consiste à "fermer les yeux" et à se mettre à la place du robot au moment où il s'exécute : tout change !
 - on ne sait pas où l'on est sur la grille et on est incapable de mesurer la distance au mur
 - il faut s'en sortir avec la seule action élémentaire disponible <Avancer>, le seul état disponible est <MurEnFace> qui peut être utilisé dans une alternative
 - énonçons une évidence : *plus on avance, plus on s'approche du mur* ; rappelons ce qui a été dit précédemment : toute action entraîne toujours une modification d'état (ici, avancer diminue la distance au mur et viendra un moment où l'exécution de <Avancer> rendra vrai l'état <MurEnFace>) ; de plus, il faut ici encore appliquer le principe de précaution : vérifier l'état avant d'exécuter l'action (cfr. situation de départ ci-dessus à droite)



on commence donc par rédiger une version plus sûre de la procédure avancer : elle permet à coup sûr d'éviter la catastrophe si le robot est déjà face au mur avant de commencer (ci-dessus à droite) ; dans ce cas : mission accomplie !

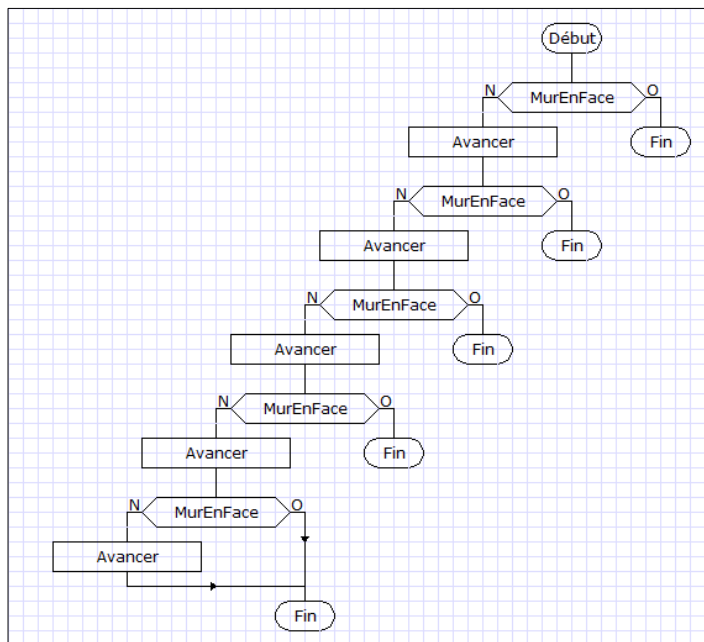
puis il faut bien utiliser cette procédure dans un algorithme principal : on pourrait évidemment rédiger ceci (ci-contre) :



... et cela fonctionne ! et ce, quelle que soit la situation de départ (position et/ou direction). MAIS on sent bien que cela ne tient pas la route : combien de fois faut-il écrire l'instruction <AvancerOuStopper> dans la séquence principale si on ne connaît pas la taille du terrain ? (imaginons qu'il fasse 1000x1000 cases et qu'il faille le traverser entièrement !)

On peut déjà remercier l'approche procédurale qui nous a évité de partir sur la piste ci-contre (pourtant équivalente quant au contenu) !

C'est le moment de se souvenir des automates-jouets ; p.ex. le lapin joueur de tambour : son mécanisme ne comporte que la séquence des actions élémentaires (gestes) à accomplir pour taper une fois sur le tambour tout en avançant d'un pas ... et ensuite ? il recommence la même chose, encore et encore ... (on comprend d'ailleurs pourquoi les mécanismes d'horlogerie sont essentiellement constitués de roues dentées ... le cercle ... principe fondamental de la répétition ...) jusqu'à épuisement de sa réserve d'énergie ; son mécanisme est conçu pour pouvoir répéter indéfiniment une même séquence

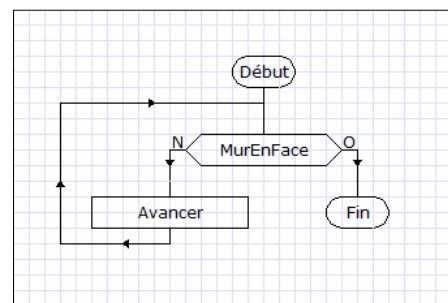


Les automates adaptables proposent plusieurs formes d'instructions de répétition (qu'on appelle plus communément en informatique des boucles) qui permettent non seulement une grande économie d'écriture, mais – on va le voir tout de suite – constituent un outil fondamental d'abstraction.

Dans notre exemple de RobotProg, on aurait évidemment envie de dire "Répète ... AvancerOuStopper ..." (mais Répète *combien de fois* ?) On ne va tout de même pas écrire "Répète 1000 fois AvancerOuStopper" ! si le robot est à 2 cases du mur, il fera 998 comparaisons inutiles et de toute façon, que se passera-t-il le jour où le terrain devient 2000x1500 cases ?

Si l'on examine attentivement la version si 'affreuse' ci-dessus, on s'aperçoit qu'il suffirait d'une simple ligne pour tout résoudre : elle partirait du 'dessous' de <Avancer> et remonterait de manière à rejoindre l'alternative : deux remarques importantes à ce stade :

- cela marche parfaitement : que la grille fasse 2x2 cases ou 1000x2000 cases et quelles que soient la position et l'orientation du robot au départ, la simple association dans le bon ordre d'une action et d'un état dans une alternative suffisent pour permettre au robot d'atteindre son but (le mur) en toute autonomie.
- MAIS : il s'agit d'une représentation graphique (appelée ordinogramme et qui était la seule utilisée jusqu'à la fin des années 1970) et cette ligne qui 'remonte' est très dérangement quand on essaie d'exprimer l'ordinogramme de manière équivalente avec le langage naturel : essayons quand même ...



AtteindreLeMur :

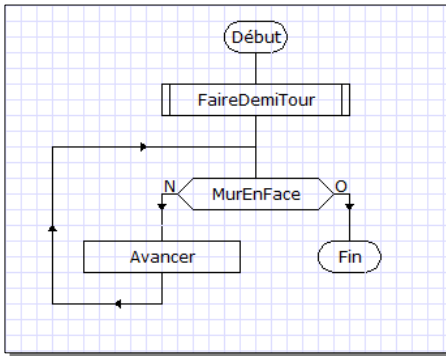
```

Début
Si non MurEnFace
Alors Avancer
    et RecommencerAprès Début
Fin
  
```

... cela peut encore se défendre, mais si la mission du robot devient :

« *quelle que soit la position du robot, atteindre le mur opposé et s'y arrêter* »

on commence par faire demi-tour (cette opération est sans risque) et grâce à la rédaction d'une procédure, le robot possède désormais cette compétence, puis on applique la méthode permettant d'atteindre le mur, mais comme dire cela en langage naturel ?



AtteindreLeMurOpposé :

Début

FaireDemiTour

Si non MurEnFace

Alors Avancer

et RecommencerAprès FaireDemiTour

Fin

Dès qu'un algorithme devient complexe, même en jouant la carte procédurale, la représentation par ordigramme révèle rapidement ses limites : des tas de boîtes et de lignes avec des flèches qui descendent, remontent, se croisent (ses détracteurs l'ont d'ailleurs baptisée 'programmation spaghetti')

A partir du milieu des années 1970 on voit émerger de nouvelles représentations pour la problématique but – méthode – action : l'algorithmique descendante utilisant le langage naturel

L'algorithmique descendante est un mode de pensée visant à construire des raisonnements en partant d'un niveau très général et en le détaillant peu à peu, jusqu'à arriver au niveau de description le plus bas.

Nous l'avons déjà utilisée avec la notion de procédure (enfermer au fur et à mesure des morceaux indépendants du problème sous forme d'une séquence nommée) et aussi avec l'alternative utilisant une condition et les mots Si, Alors et Sinon

Il reste à présent à réfléchir sur la notion de répétition et de trouver 'les mots pour le dire' ...

Une instruction de répétition définit quelle(s) instruction(s) seront exécutées à de multiples reprises, en 'boucle'; une instruction de contrôle associée permet de mettre fin à ce processus répétitif

a) La Boucle « Tant Que ... »

Que penser de ceci ...

AtteindreLeMur :

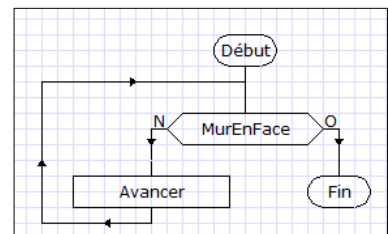
Début

Tant que non MurEnFace

Avancer

Fin

... n'est-ce pas strictement l'équivalent de ceci ?



L'instruction de répétition « Tant que » correspond au raisonnement suivant :

Tant que un but **n'est pas atteint**
continuer à **faire** ce qu'il faut
... pour atteindre **ce** but

et plus précisément, avec les outils fondamentaux (actions et états) :

Tant que un état **n'est pas vrai**
continuer à **agir**
... pour rendre vrai **cet** état

Comme on le voit en comparant la forme 'graphique' et 'linguistique', le mot «Tant que ...» correspond à la ligne qui 'remonte', et ces deux formes équivalentes comportent le même principe de précaution :

- exemple : « *quelle que soit la position du robot, atteindre le mur d'en face et s'y arrêter* »

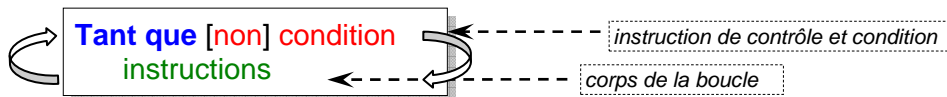
AtteindreLeMur :
Début
Tant que non MurEnFace
Avancer
Fin

Atteindre le mur, c'est rendre vrai l'état du robot <MurEnFace>, la seule action qui modifie cet état c'est <Avancer>

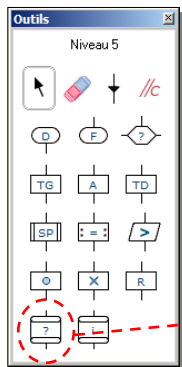
Comme la position initiale du robot est peut-être déjà face au mur, principe de précaution : il faut évaluer l'état avant de démarrer d'où la condition placée en tête, à côté de «Tant que»

L'instruction de répétition Tant que ... possède deux composantes :

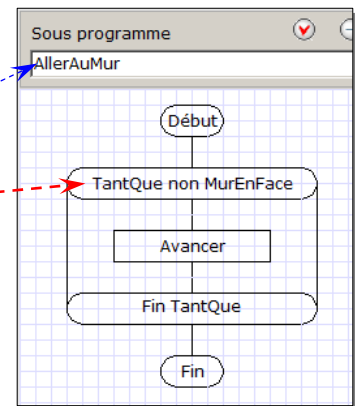
- une instruction de contrôle placée en tête permettant de décider (via une condition, vraie ou fausse) si le corps de la boucle doit être exécuté (et par conséquent si la boucle doit s'arrêter)
- un corps de boucle, comportant la ou les instructions, à (ré)exécuter et dont une au moins modifie l'état sur lequel porte la condition



Et RobotProg ? il dispose (heureusement pour nous) de l'instruction TantQue : il faut cependant passer au niveau 5 (!) pour en disposer ; elle se présente comme un bloc indépendant



- au sein duquel on place les instructions (au sens général : action élémentaire, procédure, séquence mêlant actions et procédures et même instructions de contrôle) à recommencer ;
- en tête de bloc, le mot **TantQue** est associé à (une condition sur) un état



Faisons d'une pierre deux coups : en créant une procédure **AllerAuMur** dans laquelle nous plaçons la boucle TantQue, nous donnons une nouvelle compétence aisément réutilisable et dont l'expression est simple et en parfaite adéquation avec le but recherché à nouveau, on a gagné en abstraction et en indépendance).

À remarquer que la condition sur l'état peut comporter (comme ici non MurEnFace) une négation ; si un état est vrai, alors sa négation est fausse, et réciproquement (pensez à *il pleut* et *il ne pleut pas*) ...

À retenir de la boucle TantQue : la condition placée en tête exprime en réalité une condition de continuation : on part du but à atteindre (c-à-d de la condition d'arrêt, ici MurEnFace) et en lui appliquant la négation (non MurEnFace) on indique à quelle condition le corps de la boucle doit être exécuté; après chaque exécution, la condition est réévaluée pour déterminer si "stop ou encore ..."

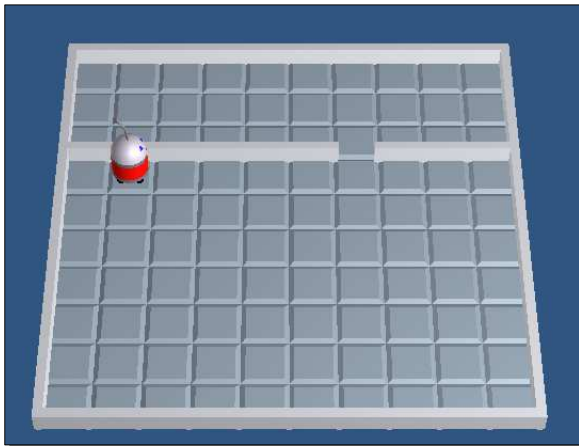
Pour s'en convaincre, il suffit d'examiner l'exemple (catastrophique) suivant :

le robot est 'au centre' du terrain (sur case non bornée sur ses quatre côtés) et la mission est : *atteindre le mur d'en face et s'y arrêter*, la procédure suivante (à droite) n'a aucune chance d'y parvenir : l'action contenue dans le corps de la boucle ne modifie pas le rapport du robot au mur et ne rendra jamais vrai l'état MurEnFace, le robot tournera indéfiniment sur place (du moins jusqu'à épuisement de sa réserve d'énergie) ... il faut donc avoir recours – comme nous l'avons fait – à l'action d'Avancer.

```

AtteindreLeMur :
Début
  Tant que non MurEnFace
    TournerADroite
  Fin
  
```

- Autre exemple : coupons le terrain en deux avec un mur muni d'une porte (ouverture) ... mission pour le robot : « passer dans l'autre pièce »



comme le problème est un rien compliqué à traiter de manière générale, on va fixer des conditions initiales (de départ) favorables en position et en direction au robot : il est 'quelque part' le long du mur intérieur (mais de toute façon pas au delà de la porte), p.ex. comme illustré

quel est le but ? *rendre vrai un état !*

quel état ? *ne plus avoir de mur à gauche !*

mais cet état n'existe pas comme tel, le seul disponible est 'MurAGauche' ! d'accord, mais il suffit de penser en termes de 'non MurAGauche'

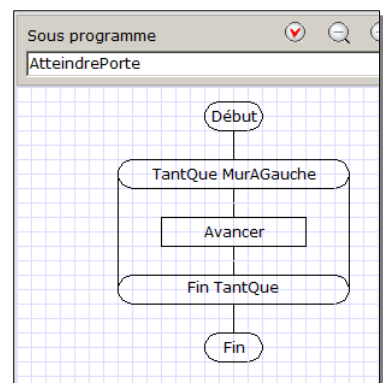
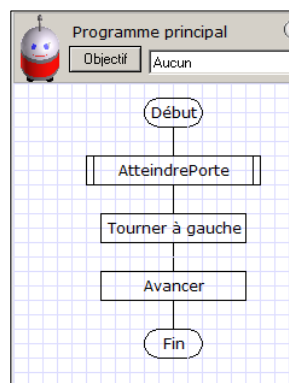
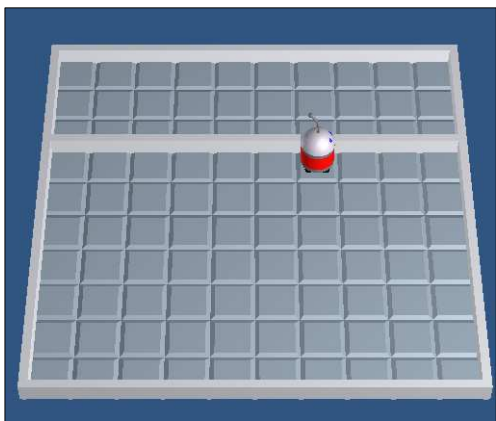
AtteindrePorte :

```

Début
  Tant que MurAGauche
    Avancer
  Fin
  
```

on voit apparaître à nouveau cette expression : tant que l'état n'est pas atteint (MurAGauche faux²²), réexécuter une action qui modifie cet état (Avancer)

et pourquoi utiliser ici le principe de précaution consistant à tester d'abord l'état avant d'avancer ? parce que le robot peut être comme ci-dessous au départ ; on a dit qu'il n'était pas au-delà de la porte, donc il peut être juste à droite de cette porte ☺



ensuite, une fois la porte atteinte, passer dans l'autre pièce est élémentaire ...

PasserDansAutrePièce :

```

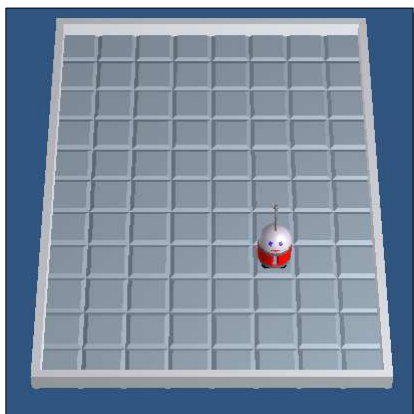
Début
  AtteindrePorte
  TournerAGauche
  Avancer
  Fin
  
```

²² donc « TantQue non MurAGauche est faux », donc TantQue MurAGauche (est vrai) ... relire encore et encore le cours de Logique ??

Encore un exemple : nouvelle mission :

« quelle que soit la position du robot, le faire patrouiller sans fin le long du mur ... »

Pour illustrer ce qui va suivre, prenons un terrain rectangulaire (8x11) et plaçons le robot au hasard ...

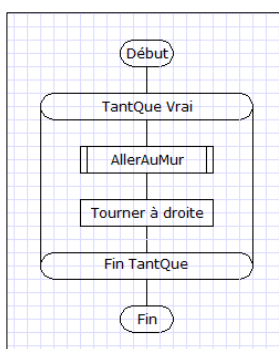
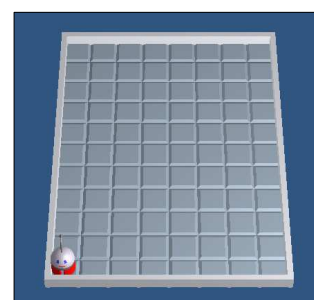


Comment lui faire faire le tour du mur d'enceinte ? ... sans doute faut-il d'abord atteindre le mur : quelle que soient sa position et sa direction, il y a toujours un mur 'quelque part' devant lui ... et on vient de lui apprendre comment l'atteindre et surtout s'y arrêter !

Ensuite, si on le fait tourner de 90° (à gauche ou à droite, peu importe) : il a à nouveau un mur 'quelque part' devant lui ... l'atteindre, s'y arrêter, tourner (dans le même sens que la fois précédente, cette fois) ... et ainsi de suite !

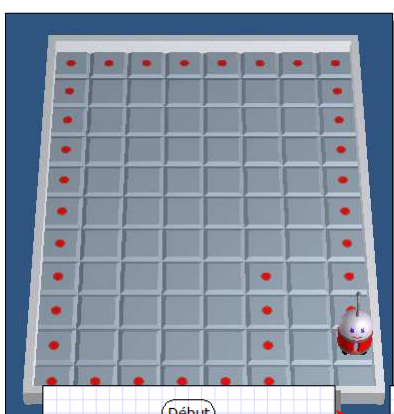
Donc on fait tout avec <AllerAuMur> et (disons) <TournerDroite>.

Si peu suffirait-il à faire tout cela ? Oui bien sûr ! et grâce au principe de précaution (condition avant action) cela marche même si le robot est déjà face au mur au départ ou même s'il est dans un coin, comme ci-contre : <MurEnFace> est vrai, donc il tourne à droite ; <MurEnFace> est à nouveau vrai, donc il tourne à nouveau à droite ... et c'est parti ... élégant, non ?

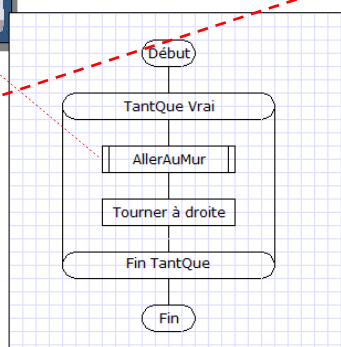
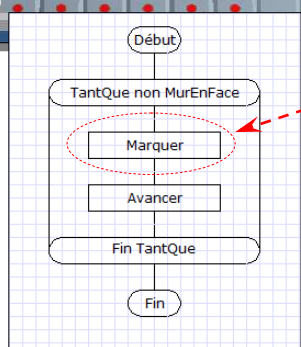
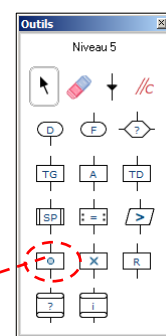


À présent, comment rédiger cette boucle sans fin ? En utilisant un TantQue avec comme condition la valeur logique Vrai (ce qui signifie : tant que vrai est vrai, donc toujours ...)

On va vérifier tout de même, et on va utiliser une nouvelle action/instruction élémentaire <Marquer> qui dépose un (gros) point rouge au sol sur la case occupée à ce moment par le robot ; cette action est associée à un nouvel état <CaseMarquée> dont on se doute



bien qu'il permettra au robot de déterminer si oui ou non il se trouve sur une case sur laquelle il y a un point rouge ... ceci sera utile pour de futures missions où le robot devra se faire 'Petit Poucet ...'²³ (bien entendu, ce que l'on a fait doit pouvoir être défait : il y a aussi une action élémentaire <Démarquer> qui enlève le point sur la case occupée par le robot (sur la palette, icône X)



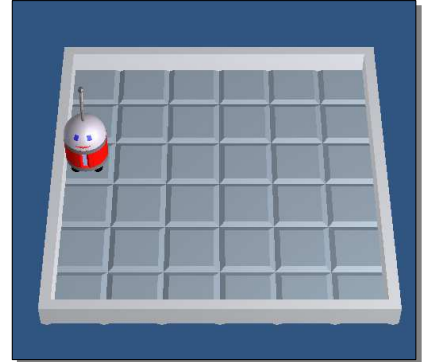
À nouveau, l'avantage de l'approche procédurale : un seul endroit à changer : la procédure <AllerAuMur> pour y ajouter l'instruction <Marquer> et cela sans aucune modification de la logique générale du problème (c.-à-d. de l'algorithme principal)

²³ à tout hasard pour qui ne connaîtrait pas : http://fr.wikipedia.org/wiki/Le_Petit_Poucet

Conservons cet exemple, mais soyons quelque peu critique : s'il est vrai que le contrat est rempli (la mission s'exécute bien, quelles que soient la position et l'orientation de départ), on remarque tout de même des situations conduisant à des comportements 'bizarres' du robot

(nous avons enregistré le dialogue entre un hypothétique couple étudiant ☺ - enseignant ☹)

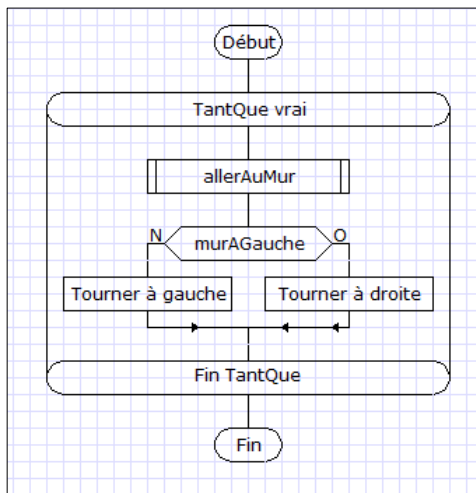
☹ par exemple, si la position et l'orientation de départ sont comme ceci, on a la surprise (?) de voir le robot faire demi-tour au prochain coin (et donc refaire une partie du chemin déjà parcouru ...), ne pourriez-vous pas rendre l'algorithme un peu plus cohérent et conserver l'orientation de départ ?



☺ ... euh ... il me semble que le problème vient du fait que l'on fait tourner le robot systématiquement à droite chaque fois qu'il est face au mur; il n'y a qu'à changer cela ...

☹ bien vu ... et que proposez-vous ?

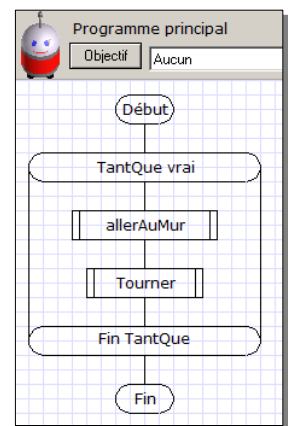
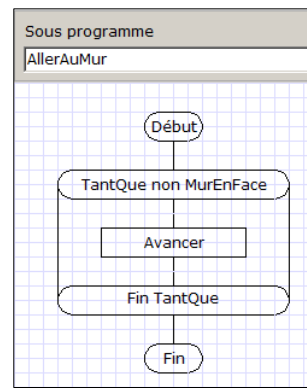
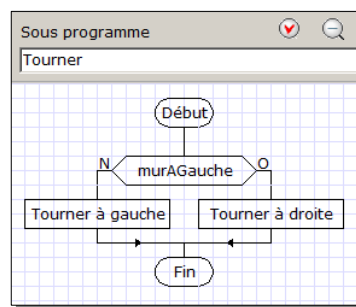
☺ dans le coin, on tourne en fonction du fait qu'on a un mur à gauche ou à droite; que pensez-vous de ceci ?



☹ intéressant, mais assez peu général, chaque fois que vous allez devoir gérer une telle situation, vous allez remettre ce test et ses deux branches dans votre algorithme ! et si l'on considérait que tourner 'dans le bon sens' devient une compétence du robot ?

☺ vous voulez que j'en fasse une procédure ? je l'appelle <Tourner> pour la rendre générale ...

voici ce que cela donne ... et cela marche parfaitement !



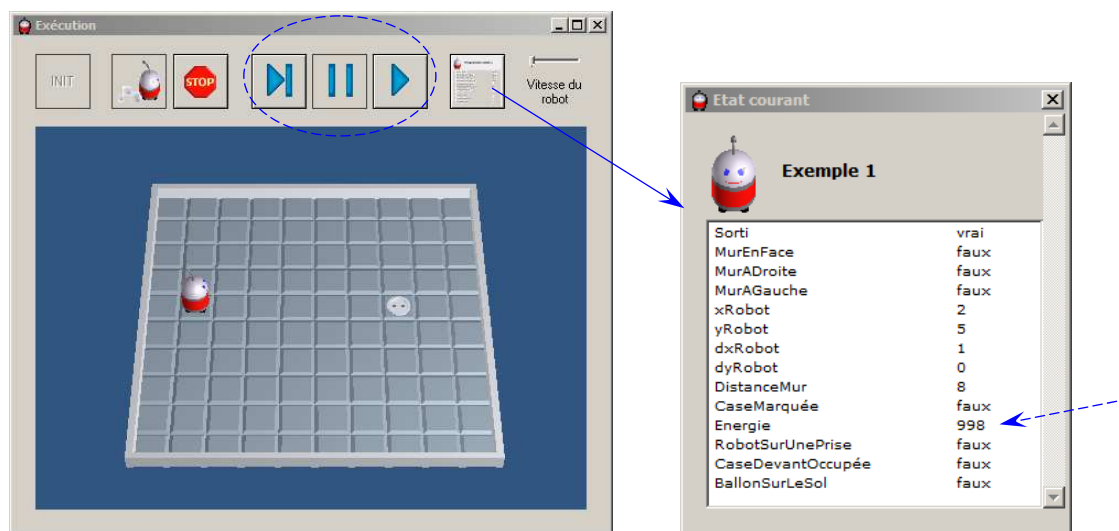
☹ pas mal en effet ... avec toutefois une petite préférence exprimée pour tourner à gauche ... si le robot n'est pas contre un mur au départ ... mais dans ce cas, on ne va tout de même pas lui demander dans quel sens il a "envie" de tourner quand il arrive face au mur !

b) Exercices

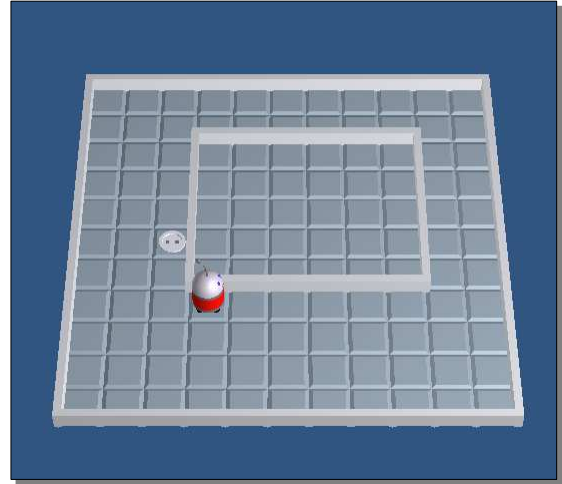
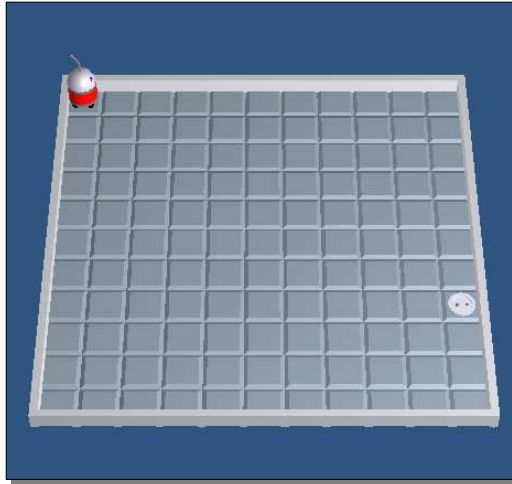
- 1) "la boucle est une version 'spéciale', plus évoluée, de l'alternative" : expliquez cette affirmation
- 2) Avec vos propres mots, expliquez ce que l'on entend par 'principe de précaution' dans le cadre des traitements 'avec boucle'
- 3) De la même manière qu'une procédure <AvancerOuStopper> a été écrite précédemment, rédiger une procédure <ReculerOuStopper>, permettant au robot de gérer son état 'dos au mur' ; bien entendu, s'il lui est impossible de reculer, le robot doit conserver sa position et sa direction d'origine !
- 4) Lors de l'exécution d'une mission, au fur et à mesure de ses déplacements (et de l'évaluation de ses états), le robot consomme de l'énergie et finit par tomber en panne. Pour éviter cette (autre) catastrophe, il est possible de placer au sol (lors de la conception ou la modification d'un terrain) des prises permettant au robot de se recharger.

Sachant qu'il existe un état <RobotSurUnePrise> et une action <Recharger>, écrire une procédure <AllerRecharger> dont les hypothèses de départ sont les suivantes : la prise est 'quelque part' devant le robot (éventuellement 'sous' lui); une fois rechargé, le robot doit s'arrêter.

NB. lors de l'exécution, il est possible de passer en mode 'pas à pas' et de contrôler les différents états du système (correspondant au mode 'debug' d'un langage de programmation)



- 5) Mettez côte à côte les algorithmes qui permettent au robot de patrouiller sans fin le long des murs, observez comment s'expriment les raisonnements ...
- dans la configuration de gauche : une prise est garantie 'quelque part' le long du mur et le robot démarre d'un coin, comme illustré (ensuite vérifiez que votre raisonnement est toujours valable si la prise est dans un coin et/ou si le robot ne part pas d'un coin ... et si le robot est sur une prise au départ ...)
 - dans la configuration de droite : idem sauf qu'il s'agit d'un mur intérieur ! quelles difficultés pourriez-vous rencontrer ici ?



6.3. PAUSE ... BASES D'UN LANGAGE ALGORITHMIQUE COHÉRENT (1)

Faisons le point pour mettre un peu d'ordre dans nos idées : nous disposons maintenant de divers outils aux rôles bien distincts :

- des actions élémentaires (p.ex. avancer, tourner à gauche, marquer ...) qui sont spécifiques à l'automate utilisé (c'est ce qu'il est capable de 'faire')
- des états logiques vrais ou faux (p.ex. mur en face, case marquée) qui sont également spécifiques à l'automate utilisé (c'est ce qu'il est capable de 'sentir')
- un langage algorithmique qui utilise des mots de la langue usuelle : *Début, Fin, Si, Alors, Sinon, Tant que, faux, vrai*, etc ... : c'est avec un tel langage que l'on construit et que l'on rédige des raisonnements structurés à l'aide de quatre types d'instructions :
 - définition de procédure (considérée comme une séquence nommée d'instructions)
 - exécution d'actions (ou de procédures)
 - prise de décision sur base d'états (Si ... Alors ... Sinon ...)
 - répétition d'actions (ou de procédures) contrôlée par la valeur d'un état logique (Tant que ...)

On s'en doute bien : si les deux premiers types d'outils (actions et états) seront différents selon l'automate à programmer (d'autres automates adaptables que RobotProg possédant leur propre jeu d'actions et d'états seront proposés un peu plus loin dans le texte), le langage algorithmique lui reste toujours 'bon pour le service' : il permet d'exprimer de manière claire, structurée et rigoureuse, utilisant un langage proche du langage naturel, la démarche de pensée allant du problème à sa solution.

... claire, structurée et rigoureuse ... cet objectif mérite qu'on s'y arrête un instant ...

Si on avait utilisé directement un langage de programmation particulier (quel qu'il soit) pour dire nos raisonnements, nous aurions été confrontés à des règles d'écriture très strictes (mots réservés, syntaxe sans pitié ...) ; ce n'est pas parce que nous privilégions le langage naturel que nous pouvons tout nous permettre (et surtout le flou, l'à-peu-près, l'imprécision, l'ambiguïté ...)

Dans un processus idéal (celui que nous essayons de poursuivre), l'algorithme est conçu d'abord et, une fois vérifié et terminé du point de vue abstrait et logique, il est finalement traduit à l'aide d'un langage de programmation ; cette opération de 'traduction' est qualifiée habituellement de codage.

Pour 'dire' les algorithmes avec le langage naturel, nous avons besoin de quelques règles spécifiques d'écriture, conduisant à un 'langage algorithmique' ; ce langage²⁴, parce qu'il est indépendant de tout langage de programmation, mais suffisamment proche tout de même pour permettre un codage aisé, sera qualifié de 'langage pseudo-code'²⁵.

Nous allons donc – sur les outils découverts jusqu'ici – mettre un rien de rigueur syntaxique, il n'est pas trop tard, mais il est temps !

²⁴ on reconnaît ici la problématique soulevée par Arzac ... comment dire les algorithmes ? : quelle que soit l'approche, il faut tout de même un langage pour pouvoir s'exprimer ...

²⁵ le langage pseudo-code proposé dans ce cours est dérivé de celui établi à l'origine par Niklaus WIRTH, lequel dans la foulée a créé le langage de programmation Pascal, ce qui explique les ressemblances assez frappantes entre l'un et l'autre. (et justifie l'adossement à ce cours d'algorithmique d'un cours de programmation utilisant le langage Pascal)

N. Wirth est à l'origine de la loi de Wirth ... à méditer (surtout l'exemple utilisé) : http://fr.wikipedia.org/wiki/Loi_de_Wirth

a) L'instruction conditionnelle (alternative) Si ... Alors ... Sinon ...

Récrivons ses deux formes (alternative 'simple' et alternative 'double') comme ceci

(les couleurs ne sont là que pour faire ressortir les composants ...)

```
Si [non] condition Alors
  instruction 1
  instruction 2
  instruction 3
  instruction 4
```

```
Si [non] condition Alors
  instruction 1
  instruction 2
  instruction 3
Sinon
  instruction 4
  instruction 5
  instruction 6
```

Que faut-il voir ?

- une présentation : les (séquences d')instructions à exécuter dans les branches de l'alternative (en vert) sont décalées à droite par rapport aux lignes qui contiennent les mots du langage : Si ... Alors ... et Sinon ... ; on dit qu'il y a indentation : c'est de la simple mise en forme mais cela fait ressortir la structure et augmente la facilité de lecture et donc la compréhension du texte ; mais cette indentation est laissée à l'appréciation du rédacteur²⁶ de l'algorithme (il peut la mettre ou non, et si oui, il a le choix de sa 'largeur')
- il y a donc un problème potentiel : l'ambiguïté : comment savoir si à gauche l'*instruction 4* fait ou non partie du Alors ... et à droite si l'*instruction 6* fait ou non partie du Sinon ... ?

Que faire ?

- tout simplement fermer l'instruction alternative Si ... Alors ... Sinon ... (en faire un bloc à part entière, de manière à pouvoir dire sans ambiguïté : l'instruction commence ici et elle se termine là) : deux méthodes vous sont proposées²⁷ :

ajouter un terminateur de fin d'instruction par exemple FinSi, ou FSi, ou FS

utiliser un marqueur de début et fin de séquence, par exemple { et }

```
Si [non] condition Alors
  instruction 1
  instruction 2
  instruction 3
FinSi
  Si [non] condition Alors
    instruction 1
    instruction 2
    instruction 3
  Sinon
    instruction 4
    instruction 5
  FinSi
  instruction 6
```

```
Si [non] condition Alors {
  instruction 1
  instruction 2
  instruction 3
}
  instruction 4
  Si [non] condition Alors {
    instruction 1
    instruction 2
    instruction 3
  }
  Sinon {
    instruction 4
    instruction 5
  }
  instruction 6
```

À remarquer : s'il existe une forme 'simple' (sans le Sinon), il n'existe pas de forme permettant de dire « Si condition *ne rien faire* Sinon *faire ceci* FinSi » (par exemple Si MurEnface Alors *rien* Sinon Avancer)

C'est ici que la négation joue un rôle essentiel : elle permet de 'renverser' les deux branches de l'alternative, et d'une certaine manière de 'convertir' un Sinon en Alors : « Si non condition Alors faire ceci FinSi » (par exemple Si non MurEnface Alors Avancer FinSi)

Note aux étudiant(e)s

Faire un choix syntaxique, c'est s'engager à s'y tenir : pas question de syntaxe floue utilisant tantôt l'une, tantôt l'autre méthode ...

²⁶ comme dans la plupart des langages de programmation : il y a quelques exceptions dont le 'récent' Python qui impose et utilise l'indentation comme élément structurel

²⁷ on remarquera que dans tous les cas le mot Alors pourrait même être omis (comme dans le langage naturel : s'il pleut (alors) je prends mon parapluie), mais il y a des traditions qui ont la vie dure, même en pseudo-code

b) L'instruction de répétition Tant que ...

Même problème ...

```
TantQue [non] condition
instruction 1
instruction 2
instruction 3
instruction 4
```

avec en plus ici la nécessité de séparer la partie conditionnelle (quand et comment s'arrêter) du corps de la boucle (ce qu'il faut ré-exécuter)

... même(s) solution(s)

ajouter un séparateur condition/corps (Faire)

*ajouter un terminateur de fin d'instruction
par exemple FinTantQue, FinTQ ou FTQ*

```
TantQue [non] condition Faire
instruction 1
instruction 2
instruction 3
FinTantQue
instruction 4
```

*utiliser un marqueur de début et fin de
séquence, par exemple { et }*

```
TantQue [non] condition {
instruction 1
instruction 2
instruction 3
}
instruction 4
```

Et à nouveau le rôle essentiel de la négation : elle permet de conserver à la condition son rôle de condition de continuation (recommencer un cycle de la boucle) selon la nature du but :

longer un mur : TantQue MurAGauche Avancer FinTQ

aller au mur : TantQue non MurEnFace Avancer FinTQ

c) Le concept d'instruction : Conclusion provisoire

Le concept d'instruction est assez difficile à définir rigoureusement (partir chercher une telle définition dans les livres spécialisés ou sur Internet, c'est à coup sûr rentrer bredouille) ... au mieux, on trouvera : « *une instruction, c'est l'expression (la spécification) de quelque chose ...* »

On a vu apparaître différents types d'instructions :

- celle qui 'fait faire' : exécution d'action, exécution de procédure ...
- celle qui 'fait décider' : l'alternative
- celle qui 'fait recommencer' : la boucle
- celle qui 'déclare' : la définition de procédure ... *elle a un statut particulier, on y reviendra*

Chacune d'entre elles offre l'expression d'une compétence particulière (donc offre une forme d'abstraction intéressante : agir – décider – refaire ...) et doit être perçue comme un bloc indépendant (possédant, on le remarquera, un début et une fin parfaitement identifiables) ; ainsi peut-on parler de l'instruction Si, de l'instruction TantQue ...

Quand on écrit :

```
TantQue condition Faire
instruction 1
instruction 2
instruction 3
FinTQ
```

il n'y a qu'une seule instruction (la boucle, qui commence au mot TantQue et qui se termine au mot FinTQ) possédant une instruction de contrôle (entre le mot TantQue et le mot Faire) et dont le corps est constitué d'une seule séquence (laquelle comporte ici trois instructions, entre le mot Faire et le délimiteur de fin FinTQ)

de même ici : une seule instruction (l'alternative, qui commence au mot Si et qui se termine au mot FinSi) avec deux branches dépendant de la condition (*à remarquer si ce n'est déjà fait qu'une seule des deux branches sera exécutée*) : chaque branche comporte une seule séquence (ici la première composée de trois instructions et la seconde de deux instructions)

```
Si [non] condition Alors
instruction 1
instruction 2
instruction 3
Sinon
instruction 4
instruction 5
FinSi
```

Ce qu'il faut bien comprendre à présent : grâce à la définition (?) de ce que sont les instructions (agir – décider – refaire), on peut mettre celles-ci partout où un tel placement est autorisé (donc dans toute séquence) et les 'mélanger' c.-à-d. les emboîter (dans le jargon algorithmique on dira les imbriquer) de manière à construire des raisonnements complexes mêlant actions, décisions et boucles ...

ainsi on pourra avoir toutes les variations possibles et imaginables, quasiment à l'infini :

```

Si condition1 Alors
  instruction 1
  Si non condition2 Alors
    instruction 2
    instruction 3
  FinSi
Sinon
  TantQue condition3 Faire
    instruction4
  FinTQ
  instruction 5
FinSi

```

```

TantQue condition1 Faire
  instruction 1
  TantQue non condition2 Faire
    instruction 2
    instruction 3
  Si condition3 Alors
    instruction 4
  Sinon
    TantQue condition 4 Faire
      instruction 5
    FinTQ
  FinSi
FinTQ
instruction 6
FinTQ

```

à remarquer au passage : le rôle de l'indentation (pour la lisibilité) et des marqueurs de fin d'instructions (pour la structure) ...

... et remarquer également que l'on aura bien besoin quand cela devient trop complexe d'enfermer certains "morceaux" dans des procédures ne serait-ce que pour augmenter la lisibilité

d) L'algorithme principal et les Procédures

Ils sont assez semblables dans leur forme : tous deux ont un nom et un contenu qui n'est rien d'autre qu'une séquence d'instructions.

Définition d'une procédure :

- annoncée par le mot 'Procédure' suivi du nom qui lui est donné (et d'un :)
- vient ensuite la séquence des instructions qui la composent placée entre deux marqueurs marquant son 'Début' et sa 'Fin'²⁸
- ... avec – de préférence - une présentation très lisible utilisant l'indentation 'ni trop, ni pas assez')

exemple :

```

Procédure AtteindrePorte :
Début
  TantQue MurAGauche Faire
    Avancer
  FinTQ
Fin

```

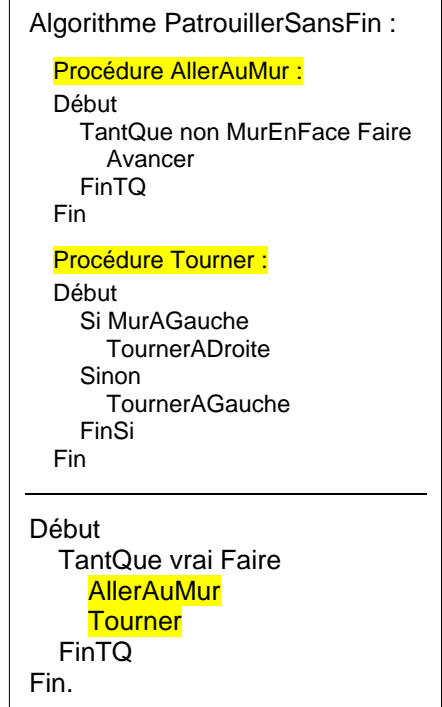
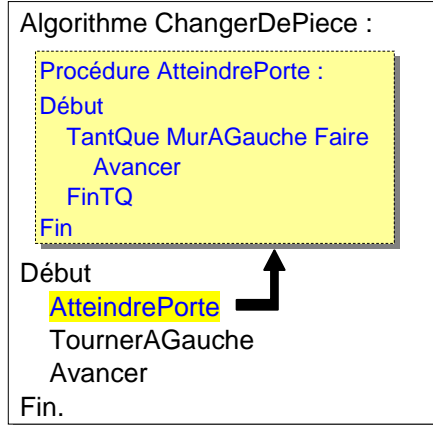
Définition d'un algorithme (procédure 'principale' ou séquence 'principale')

- annoncé par le mot 'Algorithme' suivi du nom qui lui est donné (c'est comme une 'super-procédure') (et d'un :)
- vient ensuite la séquence des instructions qui le composent placée entre deux marqueurs marquant son 'Début' et sa 'Fin'²⁹
- ... avec – de préférence - une présentation très lisible utilisant l'indentation 'ni trop, ni pas assez')
- MAIS si l'algorithme utilise dans sa séquence d'instructions une procédure, la définition de celle-ci doit figurer avant cette séquence (on ne sait pas utiliser quelque chose qu'on ne connaît pas)

²⁸ on pourrait également remplacer le couple <Début> et <Fin> par les marqueurs de séquence déjà rencontrés : { et }

²⁹ idem

exemples :



e) Les Commentaires

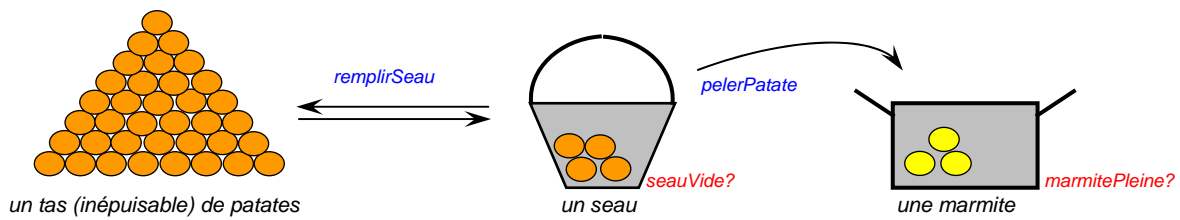
même si nos raisonnements sont rédigés en pseudo-code proche du langage naturel, il est parfois nécessaire de donner une petite explication ici ou là ... laquelle ne doit évidemment pas être interprétée comme une instruction !

les commentaires seront donc eux aussi annoncés par un symbole : ils commentent par le délimiteur # et se terminent (sans besoin de délimiteur de fin) à la fin de la ligne dans laquelle ils sont placés



6.4. EXEMPLES ET EXERCICES : D'AUTRES AUTOMATES ADAPTABLES ...

▪ Exemple : le Peleur de Patates ...³⁰



description de l'automate : il dispose d'un seau qu'il peut aller remplir à un tas de pommes de terre. Ce tas est de taille suffisante pour satisfaire n'importe quelle commande : nous dirons qu'il est inépuisable. Le robot est aussi capable de prendre une pomme de terre dans le seau, de la peler et de la jeter pelée dans la marmite.

Pour pouvoir être 'programmé', l'automate propose des actions élémentaires et des états logiques :

Actions	description
remplirSeau	prend le seau, va au tas et ramène un seau de patates
pelerPatate	prend une patate dans le seau, pèle la patate, dépose la patate pelée dans la marmite

États	description
marmitePleine	vrai quand la marmite est pleine, faux sinon
seauVide	vrai si le seau ne contient aucune patate, faux sinon

mission de l'automate : remplir la marmite de patates pelées

outils algorithmiques : ceux étudiés jusqu'à présent : instruction, séquence, procédure, contrôle par alternative (si alors sinon) et répétition (tant que)

remarques préliminaires :

- on sent bien ici le côté 'procédural caché' des actions 'élémentaires' qui nous sont proposées (il n'y a qu'à voir leur description pour percevoir une séquence d'actions plus élémentaires encore) : le robot que nous allons programmer présente un haut niveau d'abstraction (on considère que ces actions sont 'atomiques' c.-à-d. qu'elles s'exécutent complètement ou pas du tout (p.ex. s'il y a une panne de courant pendant l'exécution de <pelerPatate>, l'automate ne reste pas avec la patate 'en main')
- attention au sens des mots utilisés pour les états : (le contraire de plein n'est pas vide et réciproquement)
- on remarquera qu'il n'est fait aucunement mention des contenances respectives du seau et de la marmite, pas plus d'ailleurs que de la taille des patates : on demande d'écrire un algorithme décrivant le fonctionnement du système (algorithme abstrait) sans s'embarrasser de détails pour le moment superflus !
- de même rien n'est spécifié quant à l'état du seau au départ (vide ou pas vide) ni de la marmite au départ (pleine ou pas pleine) ; par contre, on peut considérer que le tas n'est jamais vide (inépuisable est-il dit ...)

algorithme :

- **quel est le but ?** remplir la marmite, donc rendre vrai l'état <marmitePleine>
- **quelle instruction (action) modifie cet état ?** <pelerPatate> (chaque patate pelée nous rapproche de l'état <marmitePleine>) ; puisque l'état de la marmite est inconnu, il faut appliquer le principe de précaution (elle peut être déjà pleine), donc on peut commencer à écrire :

³⁰ repris de Ch. Duchâteau, « Images pour programmer » FUNDP 1990

il semblerait cet exemple soit dû à un des 'pères' de l'algorithmique moderne : E. Dijkstra, dans « A short introduction to the art of programming » en 1971

Algorithme Peleur :

```
Début
  TantQue non marmitePleine Faire
    pelerPatate
  FinTQ
Fin.
```

- si le seau était inépuisable, ce serait parfait ... mais il faut tenir compte que l'action <pelerPatate> modifie également l'état du seau (chaque patate pelée nous rapproche aussi de l'état <seauVide> qui peut être atteint à tout moment, en particulier dès le départ si le seau est vide) ; dès lors, il faut vérifier cet état avant de peler ; dès lors on pourrait corriger l'algorithme

Algorithme Peleur :

```
Début
  TantQue non marmitePleine Faire
    Si seauVide Alors
      remplirSeau
    Sinon
      pelerPatate
    FinSi
  FinTQ
Fin.
```

- on peut vérifier que cette version fonctionne dans tous les cas de figure (les quatre³¹ combinaisons possibles des états <seauVide> et <marmitePleine>)
- c'est le moment d'améliorer l'algorithme : après avoir rempli le seau, il est dommage de ne rien faire et d'attendre le tour de boucle suivant pour peler une patate (si on va remplir le seau, c'est que la marmite n'est pas pleine ! dès lors pourquoi ne pas utiliser la séquence : remplir et puis peler ?

Algorithme Peleur :

```
Début
  TantQue non marmitePleine Faire
    Si seauVide Alors
      remplirSeau
      pelerPatate
    Sinon
      pelerPatate
    FinSi
  FinTQ
Fin.
```

- et maintenant, c'est comme en math : l'instruction <pelerPatate> figure dans la branche Alors et dans la branche Sinon : mettons-la en évidence (mais derrière !) pour rendre l'algorithme plus simple, plus lisible et plus proche du problème posé : on voit en effet disparaître la branche Sinon

Algorithme Peleur :

```
Début
  TantQue non marmitePleine Faire
    Si seauVide Alors
      remplirSeau
    FinSi
    pelerPatate
  FinTQ
Fin.
```

³¹ exercice : pourquoi quatre combinaisons et lesquelles ?

- ce que l'on peut encore écrire (de préférence uniquement si le Alors ne contient qu'une seule instruction et qu'il n'y a pas de Sinon) :

Algorithme Peleur :

Début

TantQue non marmitePleine Faire

Si seauVide Alors remplirSeau FinSi

pelerPatate

FinTQ

Fin.

Exercice : comment écririez-vous (trois versions de) l'algorithme si vous aviez la certitude que :

* la marmite n'est pas pleine au départ

* le seau n'est pas vide au départ

* la marmite n'est pas pleine et le seau n'est pas vide au départ

Exercices : missions RobotProg

sur un terrain sans obstacle de 10 x 10 cases, effectuer les missions suivantes :

consignes : *algorithmique descendante* : partir du problème général, le décomposer en sous-problèmes, dégager des 'compétences élémentaires' et en faire des procédures réutilisables

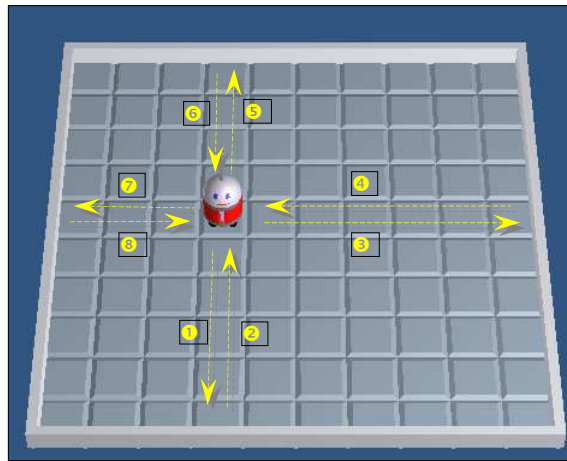
- 1) patrouiller indéfiniment (jusqu'à épuisement de la réserve d'énergie) et 'en croix' (comme illustré ci-contre) avec la position de départ comme 'pivot'

Conditions initiales :

Position du robot :	sur le terrain, mais ni contre un mur ni dans un coin
Direction du robot :	quelconque

Éléments du langage autorisés³² :

Procédures :	avancer, tournerDroite, marquerCase
États logiques :	murEnFace, caseMarquée
Instructions :	TantQue, Si Alors Sinon, Procédure



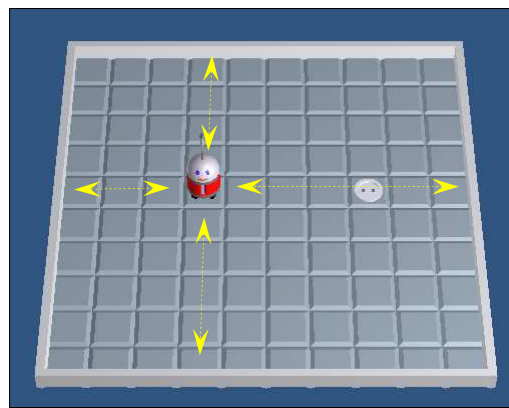
- 2) patrouiller indéfiniment (en évitant l'épuisement de la réserve d'énergie) et 'en croix' (comme précédemment) avec la position de départ comme 'pivot'. On a la certitude qu'une prise se trouve quelque part sur le trajet et que le robot pourra ainsi utiliser la procédure <Recharger> (icône R de la palette) grâce à l'état <RobotSurUnePrise>. Cet état est vrai quand le robot est sur la case qui contient une prise, il est faux dans tous les autres cas

Conditions initiales :

Position du robot :	sur le terrain, mais ni contre un mur ni dans un coin
Direction du robot :	quelconque
Position de la prise	sur le trajet du robot, mais inconnue (elle peut être contre un mur et même sous le robot au départ)

Éléments du langage autorisés :

Procédures :	avancer, tournerDroite, marquerCase, Recharger
États logiques :	murEnFace, caseMarquée, RobotSurUnePrise
Instructions :	TantQue, Si Alors Sinon, Procédure

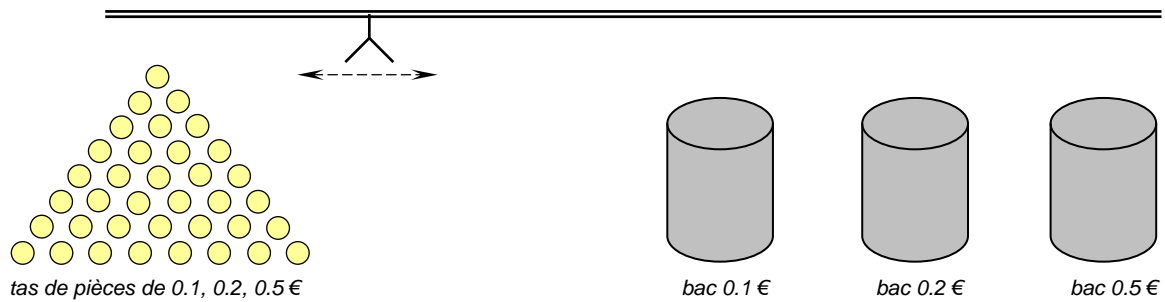


NB : il faut un minimum de corrections de l'algorithme précédent (idéalement une seule, à un seul endroit) ; sinon, c'est que l'indépendance n'a pas été une préoccupation suffisante ; il y a lieu alors de recommencer l'exercice 1) dans cet esprit avant de passer au 2)

- 3) récrire ces deux exercices en pseudo-code selon la démarche décrite au § 6.3

³² il s'agit là d'une **consigne stricte** : rien d'autre n'est autorisé !!

▪ Exercice : le Trieur de Pièces ...³³



Description de l'automate : Il dispose d'une pince articulée permettant de prendre sur un tas de pièces de monnaie (de 0.1, 0.2 et 0.5€) une pièce à la fois. Cette pince peut se déplacer horizontalement le long d'un rail. Le robot est capable de déterminer (au poids p.ex.) quel type de pièce il a dans sa pince. Il dispose de trois bacs – dont il connaît la position - pour y déposer respectivement les pièces de 0.1, 0.2 et 0.5 €.

Pour pouvoir être 'programmé', l'automate propose des actions élémentaires et des états logiques :

Actions	description
prendrePièce	aller au tas et prendre une pièce avec la pince
déposerBac1	va au bac de 0.1€ et y lâche la pièce qui est dans la pince
déposerBac2	va au bac de 0.2€ et y lâche la pièce qui est dans la pince
déposerBac5	va au bac de 0.5€ et y lâche la pièce qui est dans la pince

États	description
tasVide	vrai quand le tas ne contient plus de pièce, faux sinon
pièceDe1	vrai si la pièce dans la pince vaut 0.1€, faux sinon
pièceDe2	vrai si la pièce dans la pince vaut 0.2€, faux sinon
pièceDe5	vrai si la pièce dans la pince vaut 0.5€, faux sinon

mission de l'automate : mettre toutes les pièces du tas dans les bacs respectifs

outils algorithmiques : ceux étudiés jusqu'à présent : instruction, séquence, procédure, contrôle par alternative (si alors sinon) et répétition (tant que)

remarques préliminaires :

- ° rien n'est spécifié quant à l'état du tas au départ (vide ou pas vide) ni du nombre et de la proportion des pièces de 0.1, 0.2 et 0.5€ dans ce tas
- ° de même, l'état des bacs récepteurs est inconnu (il suffit d'ailleurs de jeter un œil sur les états proposés pour s'apercevoir que cela n'intervient pas dans le problème)

Exercices : comme écririez-vous (les 5 versions de) l'algorithme si vous aviez la certitude que :

- ° la pince est vide au départ (état du tas inconnu)
- ° la pince contient une pièce au départ (état du tas inconnu)
- ° le tas n'est pas vide au départ (état de la pince inconnu)
- ° le tas est vide au départ (état de la pince inconnu)
- ° le tas n'est pas vide et la pince est vide au départ

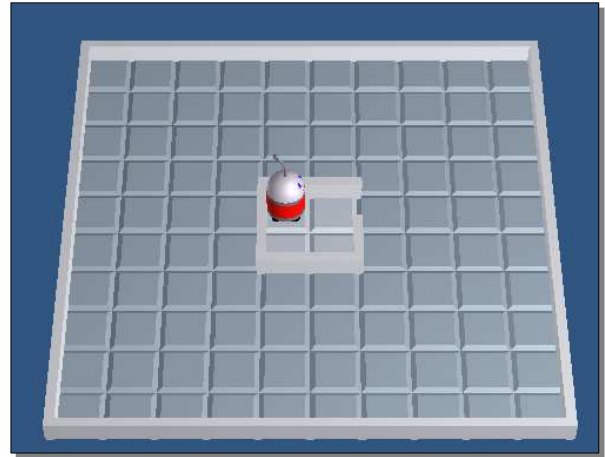
Consigne : vous devez écrire une procédure pour que votre algorithme principal soit le plus proche du problème posé

³³ repris de Ch. Duchâteau, « Images pour programmer » FUNDP 1990

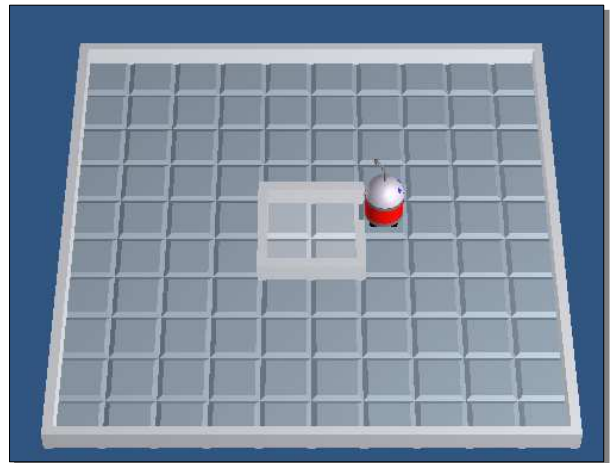
▪ Exercice : encore une mission RobotProg

Quand on démarre le logiciel RobotProg, la configuration de terrain par défaut est celle-ci : le robot est dans une 'cellule' qui ne possède qu'une seule ouverture et dont le 'pavement' est d'une couleur plus claire (cfr. outils de conception et modification d'un terrain).

À cette couleur correspond un état particulier <Sorti> qui est vrai si le robot est sur une case foncée et est faux sinon (donc dans la configuration ci-contre, la condition <Sorti> est fausse (et donc la condition non <Sorti> est vraie).



Mission : écrire un algorithme qui permette de faire sortir le robot de sa cellule et ce, quelles que soient sa position (n'importe laquelle des 4 cases) et son orientation (p.ex. face à un mur) de départ dans cette cellule ; dès qu'il est sorti, le robot doit s'arrêter (ci-contre).



Conditions initiales :

<i>Terrain</i>	comporte une cellule carrée ou rectangulaire pavée différemment du reste du terrain; cette pièce comporte une seule ouverture dont la position est indéterminée (quelconque)
<i>Position du robot :</i>	à l'intérieur de cette pièce (case inconnue, quelconque)
<i>Direction du robot :</i>	inconnue, quelconque

Éléments du langage autorisés :

<i>Procédures :</i>	avancer, tournerDroite, tournerGauche
<i>États logiques :</i>	murEnFace, murAGauche, murADroite, sorti
<i>Instructions :</i>	TantQue, Si Alors Sinon, Procédure

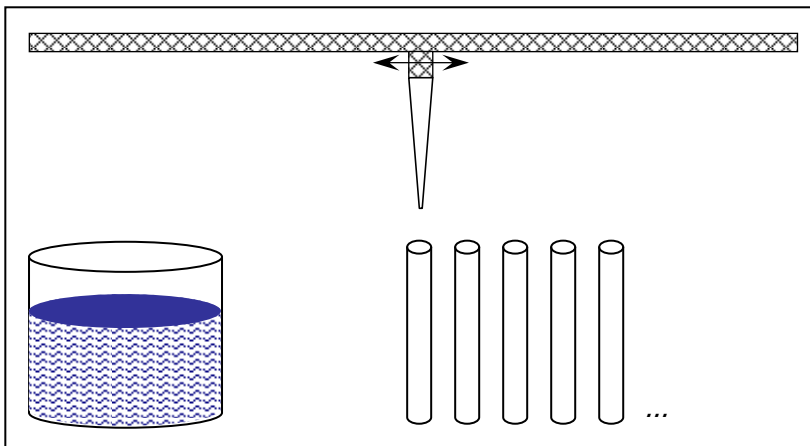
Conseil : décider d'abord comment trouner le robot : par la gauche ou par la droite et s'y tenir ...

▪ **Exercice : Un robot remplisseur d'ampoules** (*interrogation de novembre 2008*)

Une unité de production pharmaceutique dispose sous atmosphère stérile d'un robot dont le travail est de remplir automatiquement des ampoules de médicament.

Pour ce faire, il dispose d'une cuve de stockage (considérée comme inépuisable), d'une pipette accrochée à un rail et pouvant se déplacer latéralement, et d'un lot d'ampoules en nombre indéterminé.

Au démarrage, toutes les ampoules sont vides et ouvertes, la pipette est vide et est positionnée au-dessus de la première ampoule (à gauche de la série).



Le but est de remplir toutes les ampoules.

Pour commander le robot, vous disposez des procédures et des états suivants :

Procédures (actions 'élémentaires')

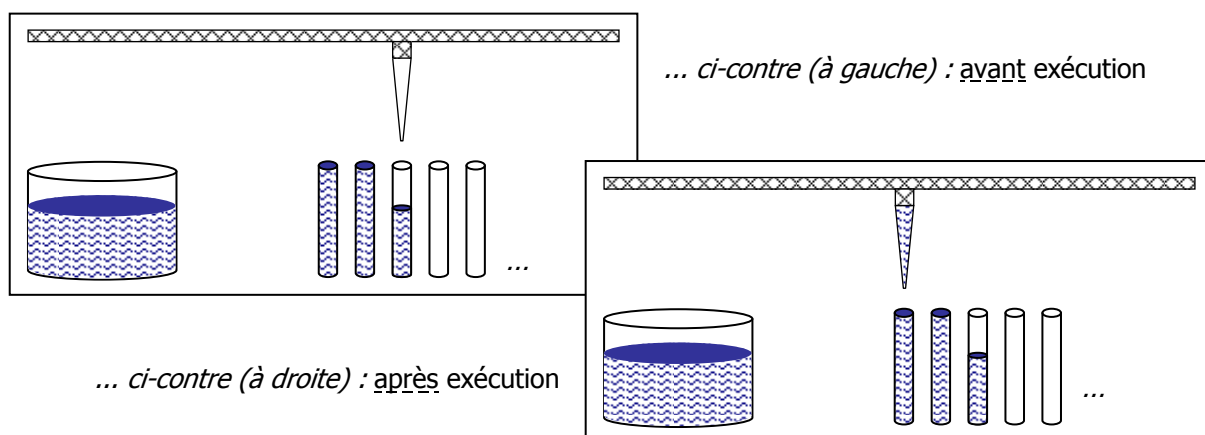
remplirPipette	Le robot quitte sa position courante, va à la cuve, remplit la pipette et se repositionne au-dessus de la <u>première</u> ampoule (à gauche)
atteindreAmpouleSuivante	Le robot déplace la pipette et la positionne au-dessus de l'ampoule suivante (de gauche à droite)
verserGoutte	Le robot presse la pipette : une goutte de médicament passe de la pipette à l'ampoule

États (logiques)

ampouleRemplie	vrai si l'ampoule correspondant à la position courante est remplie, faux sinon
pipetteVide	vrai si la pipette est vide, faux sinon
lotTerminé	vrai si <u>toutes</u> les ampoules sont remplies, faux sinon

NB : les ampoules et la pipette contiennent un nombre entier de gouttes, mais on ne sait rien quant à la contenance relative ampoule/pipette

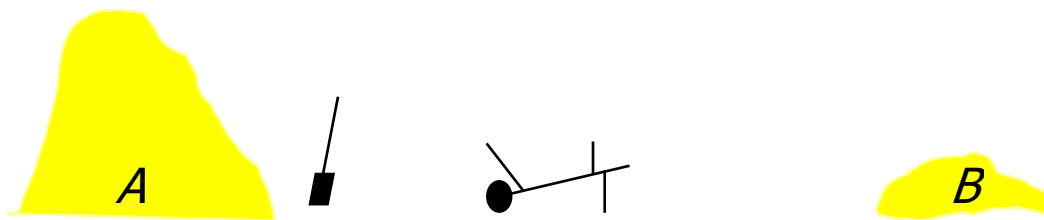
NNB : illustration de l'exécution de la procédure <remplirPipette> ...



Rédiger en pseudo-code un algorithme abstrait permettant au robot de remplir toutes les ampoules ; il est recommandé d'écrire de nouvelles procédures ...

- Exercice : Un robot transporteur de sable³⁴ (*interro de novembre 2009*)

mission de l'automate : déplacer tout le sable du tas A vers le tas B



Pour commander le robot, vous disposez des procédures et des états suivants :

Procédures

<i>pelleteoSable</i>	le robot met <u>une</u> pelle de sable dans la brouette
<i>amenerBrouette</i>	le robot déplace la brouette de A vers B
<i>déverserBrouette</i>	le robot vide la brouette en B
<i>ramenerBrouette</i>	le robot déplace la brouette de B vers A

États logiques

<i>tasVide</i>	vrai/faux	le tas A est vide
<i>brouettePleine</i>	vrai/faux	la brouette est pleine

Conditions initiales (à respecter) :

- la brouette est vide, positionnée en A
- l'état <tasVide> est faux
- !!! la contenance du tas et de la brouette sont un nombre entier de pelles de sable, mais la contenance du tas n'est pas nécessairement un nombre entier de brouettes !!

Rédigez en pseudo-code un algorithme abstrait permettant de déplacer tout le sable de A vers B.

³⁴ repris de Ch. Duchâteau, « Images pour programmer » FUNDP 1990

▪ Exercice : Un robot serveur de bière (*interro de novembre 2010*)

Dans le cadre des futures festivités "24 heures vélo de LLN 2011", vous participez à un groupe de travail pluridisciplinaire (électromécaniciens, électroniciens, informaticiens ...) chargé de réaliser un robot-automate "serveur de bières".

L'ambiance et quelques verres aidant, le groupe commence à imaginer des caractéristiques de fonctionnement de l'automate :

- il aurait à sa disposition
 - en entrée des bacs (traités un par un, à la suite) pouvant contenir aussi bien des vidanges que des bouteilles pleines et capsulées
 - en sortie des bacs vides pour y déposer les vidanges
- il devrait pouvoir déterminer
 - si le bac d'entrée est vide (il le ferait alors passer du côté 'sortie' et ferait entrer un autre bac)
 - si le bac de sortie est plein (il l'évacuerait alors et le remplacerait par un bac vide)
- il prendrait une bouteille dans le bac d'entrée et selon son état (celui de la bouteille ☺) :
 - si elle est vide, la déposerait telle quelle dans le bac de sortie
 - si elle est pleine, la servirait
- pour servir une bouteille, l'automate devrait d'abord la décapsuler, puis
 - s'il disposait de verres (vides et propres), il verserait le contenu de la bouteille dans un verre, donnerait celui-ci au client et déposerait la vidange dans le bac de sortie
 - si aucun verre n'était disponible, il donnerait directement la bouteille décapsulée au client
- la collecte et la mise à disposition de verres vides (et propres) serait le travail d'un autre automate non décrit ici

... et quelques autres verres plus tard ... le groupe de réflexion établit une première liste d'actions procédurales et d'états binaires de l'automate :

Actions

entrerBac	prendre le bac suivant en entrée pour le traiter
sortirBac	faire passer le bac d'entrée vide du côté sortie
évacuerBac	évacuer le bac de sortie plein de vidanges et le remplacer par un bac vide
prendreBouteille	prendre une bouteille dans le bac d'entrée
décapsulerBouteille	décapsuler une bouteille pleine
verserBiére	verser le contenu de la bouteille dans un verre
donnerVerre	donner le verre rempli au client
donnerBouteille	donner la bouteille décapsulée au client
déposerVidange	mettre la bouteille vide dans le bac de sortie

États binaires

bacVide	le bac d'entrée est vide
bacPlein	le bac de sortie est plein de vidanges
bouteilleVide	la bouteille est vide
verreDisponible	il y a au moins un verre vide et propre à disposition de l'automate

Sur base de cet ensemble de composants algorithmiques, rédigez une première version d'un l'algorithme fonctionnel (abstrait) de cet automate

Cet ensemble vous semble-t-il complet (si non, qu'ajouteriez-vous ?) ou trop chargé (si oui, qu'enlèveriez-vous ?)

▪ Exercice : Un robot collecteur de vidanges (*interro de novembre 2011*)

La région wallonne s'est dotée de nouvelles "bulles à verre" qui permettent la collecte de bouteilles vides tout en en garantissant d'en conserver la majorité intactes. En tant que stagiaire, vous participez à un groupe de travail pluridisciplinaire (électromécaniciens, électroniciens, informaticiens ...) chargé de réaliser un robot-automate pour le pré-traitement du recyclage de ces bulles : le but est de ne fournir que des bouteilles de verre blanc, intactes et nettoyées

À grand renfort de café(s), le groupe commence à imaginer des caractéristiques de fonctionnement de l'automate :

- il aurait à sa disposition
 - en entrée : des bulles à verre blanc (traitées une par une, à la suite), mais pouvant contenir aussi bien des bouteilles en verre 'blanc' que des bouteilles en verre 'coloré' (les consommateurs peuvent souvent se montrer peu respectueux des consignes; ils peuvent également être distraits ...); le contenu d'une bulle est pris en charge par un dispositif qui en place le contenu (les bouteilles) à la queue-leu-leu sur un tapis roulant
 - en sortie : des camions pouvant transporter des palettes de casiers de bouteilles intactes et propres à destination de l'usine de retraitement final
- il devrait pouvoir déterminer
 - s'il y a une bulle à traiter à l'entrée de la chaîne
 - s'il y a un camion disponible en sortie dans lequel il reste de la place
 - la 'couleur' de la bouteille (verre blanc ou non)
 - la 'qualité' de la bouteille (présence de défauts, bouteille cassée, fendue, etc ...)
 - le caractère rempli ou non d'un casier; même chose pour une palette (de casiers) et même chose pour un camion (de palettes)
- chaque bouteille sur le tapis roulant sera traitée à son tour (elle est alors dénommée 'bouteille courante') :
 - elle passe d'abord devant un dispositif optique qui détermine si elle est 'colorée'; si elle l'est, elle est écartée en direction d'un conteneur qui sera acheminé - une fois rempli - vers une chaîne de traitement similaire du verre coloré
 - si elle est 'blanche', elle passe devant un autre dispositif optique qui détermine son caractère 'intact' ou non; selon cet état, elle est dirigée vers un conteneur de verre cassé ou vers la suite de la chaîne : l'unité de rinçage, après quoi elle est placée dans un casier
- un casier rempli de bouteilles propres est placé sur une palette et un casier vide (si disponible) est placé sur la chaîne
- lorsqu'une palette contient son quota de casiers, elle est emballée dans du plastique et emmenée vers le camion; une palette vide (si disponible) est alors placée sur la chaîne
- un camion rempli de palettes reçoit automatiquement un 'feu vert' de départ, et un camion vide (si disponible) prend sa place en fin de chaîne de traitement

... en fin de journée, le groupe de travail propose un ensemble d'actions élémentaires et d'états binaires pour le robot

Actions

entrerBulle	<i>prendre la bulle suivante en entrée pour la traiter</i>
écarterBouteille	<i>envoyer la bouteille courante vers le conteneur de verre coloré</i>
écacuerBouteille	<i>envoyer la bouteille courante vers le conteneur de verre cassé</i>
nettoyerBouteille	<i>nettoyer la bouteille courante</i>
sortirCasier	<i>placer un casier rempli sur la palette et le remplacer par un casier vide</i>
sortirPalette	<i>placer une palette remplie de casiers dans le camion et la remplacer par une palette vide</i>
emballerPalette	<i>emballer la palette dans du plastique avant de la placer dans le camion</i>

États binaires

bulleDisponible	<i>vrai s'il y a au moins une bulle à l'entrée de la chaîne, faux sinon</i>
tapisVide	<i>vrai si le tapis roulant alimentant la chaîne ne comporte plus de bouteille, faux sinon</i>
camionRempli	<i>vrai si le camion en fin de chaîne est rempli de palette et doit partir, faux sinon</i>
verreBlanc	<i>vrai si la bouteille est en verre 'blanc' (non coloré), faux sinon</i>
défautBouteille	<i>vrai si la bouteille est fendue, cassée, ... et ne peut être traitée, faux sinon</i>
casierRempli	<i>vrai si le casier de sortie est rempli de bouteille, faux sinon</i>
paletteRemplie	<i>vrai si la palette est remplie de casiers, faux sinon</i>

Sur base de cet ensemble de composants algorithmiques, rédigez une première version d'un algorithme abstrait (comportemental) de cet automate afin de faire partir si possible un premier camion de palettes

les conditions initiales à prendre en compte sont les suivantes :

- il y a au moins un camion vide en attente à la sortie
- il y a suffisamment de bulles en attente (donc potentiellement de bouteilles) à l'entrée pour remplir ce camion
- compte tenu de la présence de bouteilles colorées et/ou de bouteilles présentant des défauts, on ne sait évidemment rien garantir quant au nombre de bouteilles 'utiles' par bulle
- on dispose sur la chaîne du nombre nécessaire de casiers vides (tous identiques) et de palettes vides (toutes identiques)

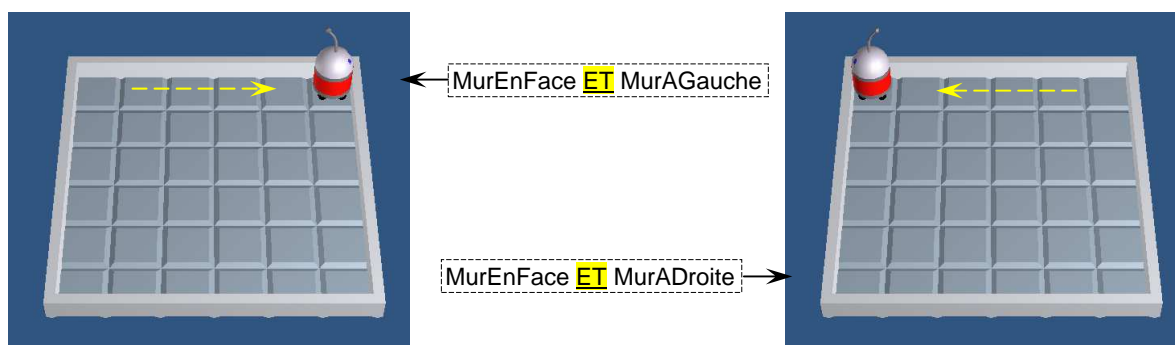
Cet ensemble vous semble-t-il complet (si non, qu'ajouteriez-vous ?) ou trop chargé (si oui, qu'enlèveriez-vous ?)

6.5. LES CONDITIONS MULTIPLES

a) Le Principe ...

Revenons sur le concept d'état d'un automate : il ne faut pas perdre de vue que si cet automate propose différents états, il est en permanence (à tout moment) 'dans tous ses états', chacun d'entre eux pouvant être individuellement vrai ou faux.

Nous allons à présent nous intéresser à la combinaison d'états (par exemple, quand Robot-Prog arrive dans un coin (en avançant) : deux états sont vrais simultanément et cela en fonction du sens de la marche



La logique³⁵ nous indique que des propositions simples (équivalentes à nos états) peuvent être combinées à l'aide d'opérateurs spécifiques (opérateurs logiques) ET et OU, et que de telles combinaisons sont elles-mêmes des propositions qui sont dès lors vraies ou fausses

Avec deux états, on a ainsi les (4) possibilités suivantes synthétisées dans les 'tables de vérité' des opérateurs logiques ET / OU (F=faux, V=vrai) ...

... sans oublier la négation (opérateur NON) qui joue ici un rôle essentiel (pour rappel, les lois de De Morgan) :

p	q	p ET q
F	F	F
F	V	F
V	F	F
V	V	V

p	q	p OU q
F	F	F
F	V	V
V	F	V
V	V	V

$$\text{NON } (p \text{ ET } q) \Leftrightarrow (\text{NON } p) \text{ OU } (\text{NON } q)$$

$$\text{NON } (p \text{ OU } q) \Leftrightarrow (\text{NON } p) \text{ ET } (\text{NON } q)$$

Exemples plus concrets :

- être dans le coin supérieur droit \equiv MurEnFace ET MurADroite (les deux vrais en même temps) et ne pas être dans ce coin \equiv NON (MurEnFace ET MurADroite) \equiv NON MurEnFace OU NON MurADroite (séparément ou simultanément)
- être dans un coin en toute généralité s'écrit en combinant trois états, par exemple : (MurEnFace ET MurAGauche) OU (MurEnFace ET MurADroite)

ce qui peut se simplifier en MurEnFace ET (MurAGauche OU MurADroite) c.-à-d. :

"de toute façon face au mur et, soit un mur à gauche, soit un mur à droite"

Exercice : pour chacun des quatre coins du terrain, faire l'inventaire des positions possibles du robot ; pour chacune, quels sont les états qui sont vrais simultanément ;

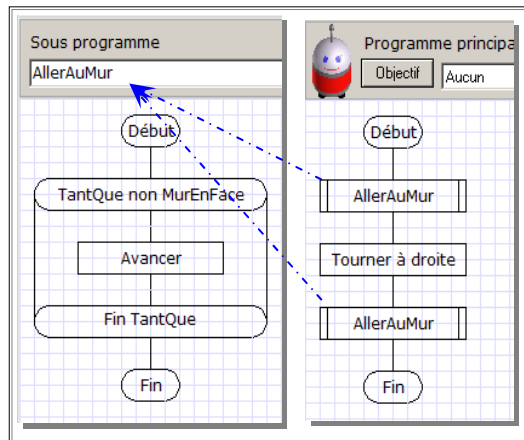
y a-t-il des positions (de coin) où un seul état est vrai et pourquoi ?

³⁵ au risque de nous répéter : le fascicule 'Éléments de Logique' dans le cours de Maths, etc , etc ...

b) ... un exemple RobotProg

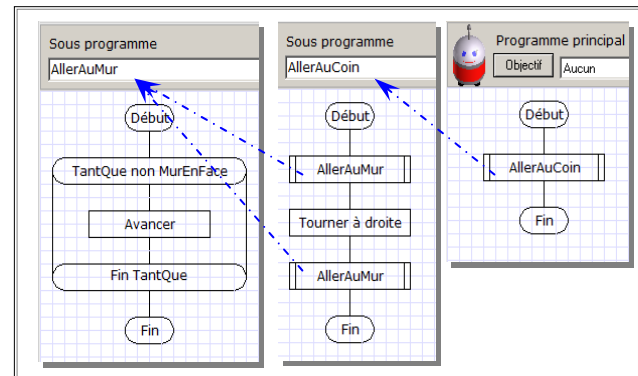
Avec ce qui vient d'être dit, « *Aller dans un coin* » semble être un bel exemple de nouvelle mission : (extrait d'un dialogue entre un hypothétique couple étudiant ☺ - enseignant ☹)

☺ allez, je me lance ... que dites-vous de ceci (j'ai fait une procédure !) ? :



☹ **bof !** où apparaît le but (l'abstraction) recherché : <AllerAuCoin> ???

☺ oups ... et comme ceci ?

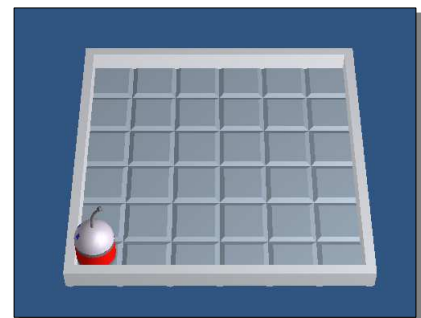


☹ en tous cas, comme ça je veux bien commencer à discuter sur le contenu de votre procédure <AllerAuCoin> : mais est-ce vraiment cela aller dans un coin ?

☺ pourtant cela marche !

☹ en êtes-vous vraiment sûr ? ... cela marche uniquement si vous choisissez les conditions de départ qui sont favorables à ce que vous avez écrit !

☹ que pensez-vous de cette situation de départ (ci-contre) : le robot est manifestement dans un coin, mais il va filer dans un autre ! (un automate n'a pas – encore - l'autonomie suffisante pour décider de faire autre chose que ce qui lui a été demandé, tout de même !)



☺ (silence embarrassé)

☹ tout simplement, vous avez oublié de penser en termes de but à atteindre et d'exprimer ce but en termes d'états ! et si on reprenait dans ce sens ? ☹ quel est le but ?

☺ être dans le coin, c.-à-d. rendre vrai l'état : <MurEnFace> ET <MurAGauche> !

☹ arrêtez de fixer obstinément la situation de départ qui vous est proposée; fermez les yeux, c'est quoi en toute généralité être dans un coin (en termes d'états) ?

☺ c'est rendre vrai l'état : <MurEnFace> ET (<MurAGauche> OU <MurADroite>)

☹ on y arrive ... appliquez à présent le principe : "tant que le but n'est pas atteint, faire ce qu'il faut pour atteindre le but (rendre vrai l'état)" ... en écrivant "tant que l'état n'est pas vrai ..."; et qu'est-ce qui peut modifier ces états ?

☺ <Avancer> et <Tourner ...>, donc j'écris ...

```
TantQue non (MurEnFace ET (MurAGauche OU MurADroite)) Faire
  Avancer
FinTQ
```

☹ ce n'est déjà pas si mal : si le robot au départ :

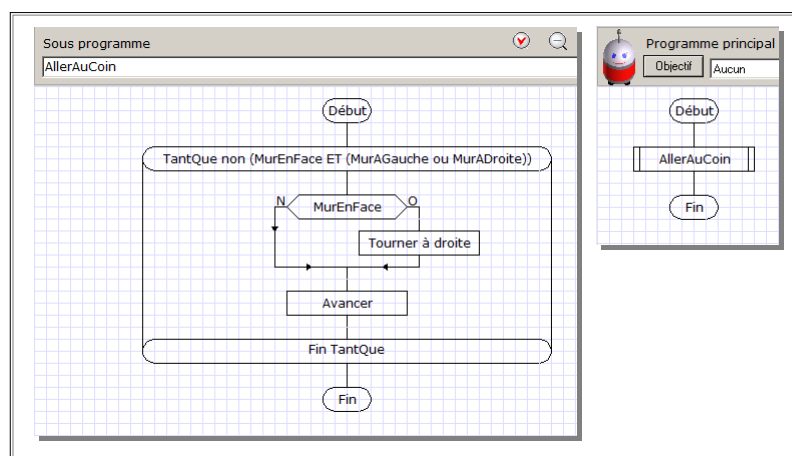
- est dans un coin, il ne fait rien, mission accomplie
- longe un mur (n'importe lequel), il va avancer et s'arrêter dans le coin 'devant' lui, mission accomplie

⊗ *MAIS s'il part du 'milieu' du terrain : il va se fracasser sur le mur d'en face : en effet vous n'avez pas prévu que l'état <MurEnFace> puisse être vrai tout seul ! vous devez donc ajouter une condition pour gérer cette situation : "si face au mur, sans mur à gauche ni à droite, alors tourner " (à gauche ou à droite à votre meilleure convenance ; et comme il peut être face au mur dès le départ, principe de précaution oblige, vous devez faire ce test avant toute tentative d'avancer ...*

☺ je dois reconnaître que vous avez raison ... donc j'écris ...

```
TantQue non (MurEnFace ET (MurAGauche OU MurADroite)) Faire
  Si MurEnFace Alors
    TournerDroite
  FinSi
  Avancer
FinTQ
```

☺ et donc en RobotProg, je réécris <AllerAuCoin> et je supprime <AllerAuMur> ...



⊗ *cela me semble bon ; si on l'essaie dans toutes les configurations de départ possibles et imaginables, on voit bien ce robot tourner la tête à droite et à gauche à chaque pas, parce que chaque pas modifie potentiellement un ou plusieurs des états intervenant dans la condition*

Cette logique n'est pas valable uniquement pour RobotProg, comparons avec ce qui a été écrit pour l'automate Peleur :

```
TantQue non marmitePleine Faire
  Si seauVide Alors
    remplirSeau
  FinSi
  pelerPatate
FinTQ
```

```
TantQue non (MurEnFace ET (MurAGauche OU MurADroite)) Faire
  Si MurEnFace Alors
    TournerDroite
  FinSi
  Avancer
FinTQ
```

étonnant, non ?

NB : la condition multiple (et en particulier la double condition OU) est un 'modèle' d'algorithme que l'on rencontre très souvent.

on le croiera à nouveau plus tard, par exemple lors de la recherche d'un élément d'une liste ... quand s'arrête-t-on de chercher ? soit parce que l'on a trouvé, soit parce que l'on a atteint la fin de la liste ; la condition d'arrêt s'écrit donc (trouvé OU finDeListe) et la forme générale de la boucle (avec condition de continuation via la négation) s'écrit donc :

```
...
TantQue non (trouvé ou finDeListe) Faire
  continuerAChercher
FinTQ
...
```

c) ... un autre exemple sur un autre automate

L'automate Peleur, justement, (re)parlons-en ... cette hypothèse d'un tas de patates inépuisable semble peu réaliste, donc supprimons-la : cela veut dire que l'automate propose toujours les mêmes procédures³⁶ mais avec un état logique supplémentaire, appelons-le <tasVide>

Procédures	description
remplirSeau	prend le seau, va au tas et ramène un seau de patates
pelerPatate	prend une patate dans le seau, pèle la patate, dépose la patate pelée dans la marmite

États	description
marmitePleine	vrai quand la marmite est pleine, faux sinon
seauVide	vrai si le seau ne contient aucune patate, faux sinon
tasVide	vrai s'il n'y a plus de patate sur le tas, faux sinon

Par contre nous conservons toutes les autres 'inconnues' de départ (taille relative seau/marmite, état du seau, état de la marmite) auxquelles nous ajoutons bien entendu la nouvelle inconnue : l'état du tas au départ et également la taille relative tas/marmite : rien ne dit qu'il y a au départ assez de patates pour remplir la marmite !

Qu'est-ce que cela change à notre raisonnement ? que chaque exécution de <remplirSeau> modifie l'état du tas, et qu'arrive (peut-être) un moment où <tasVide> devient vrai.

Qu'est-ce que cela change à l'algorithme que nous avons écrit ?

Algorithme Peleur :

Début

TantQue non marmitePleine Faire

Si seauVide Alors remplirSeau FinSi

pelerPatate

FinTQ

Fin.

Essayons-le dans les nouvelles conditions en supposant au départ

- que le tas est vide,
- que la marmite n'est pas pleine,
- que la taille du seau est plus petite que celle de la marmite
- et qu'il y a quelques patates dans le seau

la boucle principale commence à 'tourner' ...

... vient un moment où le seau est vide (et la marmite toujours non pleine) :

... on va donc aller au tas et remplir 'de vide' le seau (il n'y a pas de patate sur le tas), puis le mécanisme va peler 'dans le vide' (il n'y a rien dans le seau) ... et on a une boucle sans fin du seau qui va indéfiniment aller au tas ... pour ne pas se remplir ...

Il faut donc introduire une modification de l'algorithme : une transformation subtile du but original : expliquons-nous ...

- avant il fallait rendre vrai le seul état <marmitePleine> (et pour cela on exécutait <pelerPatate>)
- à présent il faut se poser la question : quand l'automate s'arrête-t-il ? (c'est en quelque sorte le nouveau but à atteindre, <marmitePleine> restant bien entendu le but principal) et on voit apparaître trois possibilités :
 - soit <marmitePleine> est vrai,
 - soit <tasVide> est vrai,
 - soit les deux ensemble, évidemment

³⁶ désormais nous n'utiliserons plus le terme d'action élémentaire mais uniquement sa généralisation : la procédure

- il faut nous donc baser notre nouvel algorithme sur cette double condition :

arrêt si <marmitePleine> OU <tasVide>

dans un premier temps, puisque la boucle TantQue est basée sur la continuation, nous pourrions nous contenter de mettre un non devant la nouvelle condition d'arrêt :

```

Algorithme Peleur :
Début
  TantQue non (marmitePleine OU tasVide) Faire
    Si seauVide Alors remplirSeau FinSi  # change état du tas
    pelerPatate                          # change état du seau et de la marmite
  FinTQ
Fin.

```

Mais c'est conclure un peu vite ! En effet, nous avons mis sur le même plan deux conditions d'arrêt en oubliant que l'une était principale (<marmitePleine>) et l'autre secondaire (<tasVide>) :

après avoir rempli le seau, il se peut que le tas soit vide mais arrêter la boucle pour cette seule raison ne permet peut-être pas d'atteindre la vraie mission alors que celle-ci est encore réalisable (et si les patates dans le seau suffisaient à remplir la marmite ?)

on doit donc – après avoir rempli le seau – essayer de remplir la marmite, selon la logique suivante ...

```

Algorithme Peleur :
Début
  TantQue non (marmitePleine ou tasVide) Faire  # deux conditions d'arrêt
    Si seauVide Alors remplirSeau FinSi          # change état du tas
    TantQue non (marmitePleine ou seauVide) Faire  # deux conditions d'arrêt
      pelerPatate                                # change état marmite et seau
    FinTQ
  FinTQ
Fin.

```

... qui fait apparaître – si l'on est un peu puriste – une mission dans la mission, et donc une décomposition du problème général à l'aide d'une procédure; on remarquera enfin (dans la séquence principale) que si l'on veut déterminer pourquoi la boucle s'est arrêtée, il faut faire un test explicite sur un des états intervenant dans la condition

```

Algorithme Peleur :
  Procédure viderSeau :
    Début
      TantQue non (marmitePleine ou seauVide) Faire  # 2 états : 2 conditions
        pelerPatate                                # change 2 états
      FinTQ
    Fin
  Début
    TantQue non (marmitePleine ou tasVide) Faire
      Si seauVide Alors remplirSeau FinSi  # change état du tas
      viderSeau                            # change état de la marmite
    FinTQ
    Si marmitePleine Alors
      écrire('marmite remplie !')
    Sinon
      écrire('pas assez de patates sur le tas pour remplir la marmite !')
    FinSi
  Fin.

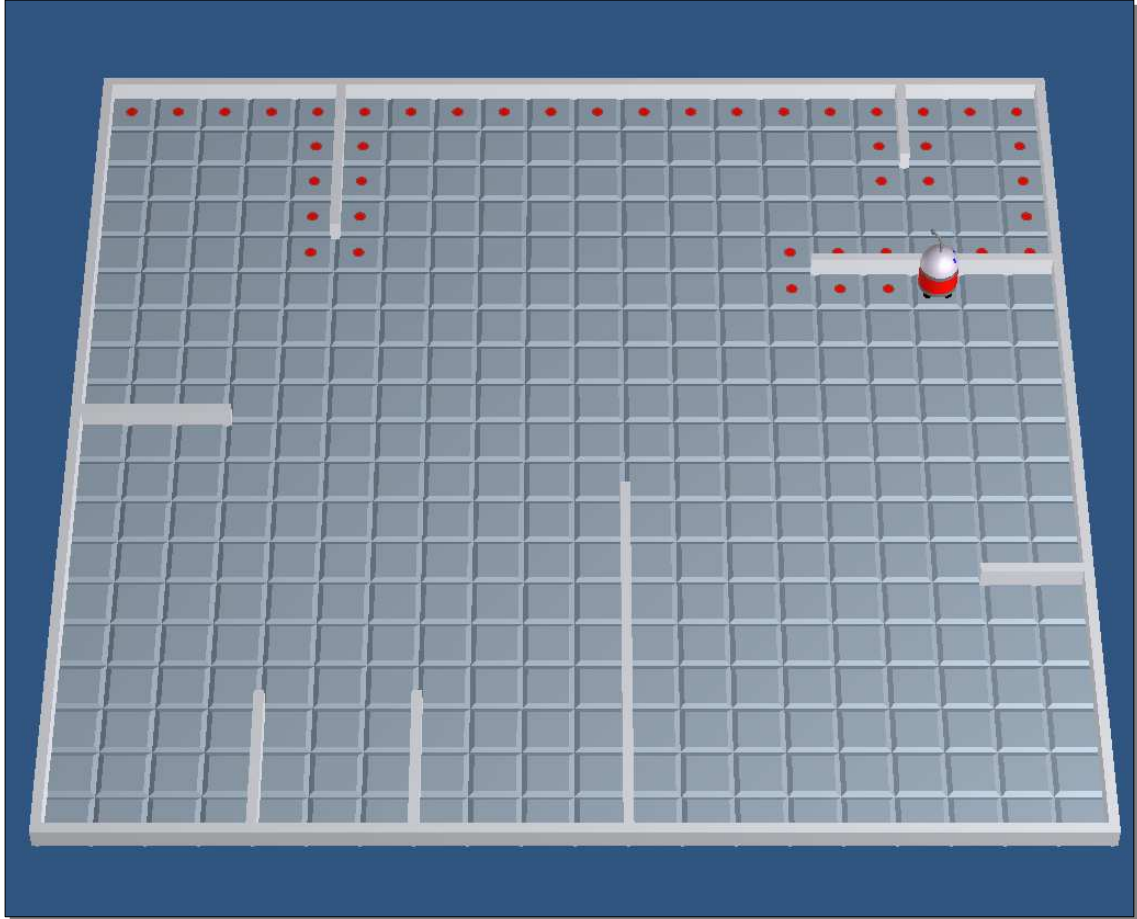
```

d) Exercices...

- Une mission RobotProg (*interro de novembre 2008*)

Objectif à atteindre :

Faire une fois le tour complet du terrain, en longeant tous les murs



Conditions initiales :

<i>Terrain :</i>	des murs droits perpendiculaires aux murs extérieurs et de longueurs quelconques
<i>Position du robot :</i>	coin supérieur gauche (figure ci-dessus)
<i>Direction du robot :</i>	sens horlogique

Éléments du langage autorisés :

<i>Procédures :</i>	avancer, tournerGauche, tournerDroite, marquerCase
<i>États logiques :</i>	murEnFace, murAGauche, murADroite, caseMarquée
<i>Langage algorithmique :</i>	TantQue, Si Alors Sinon, Procédure

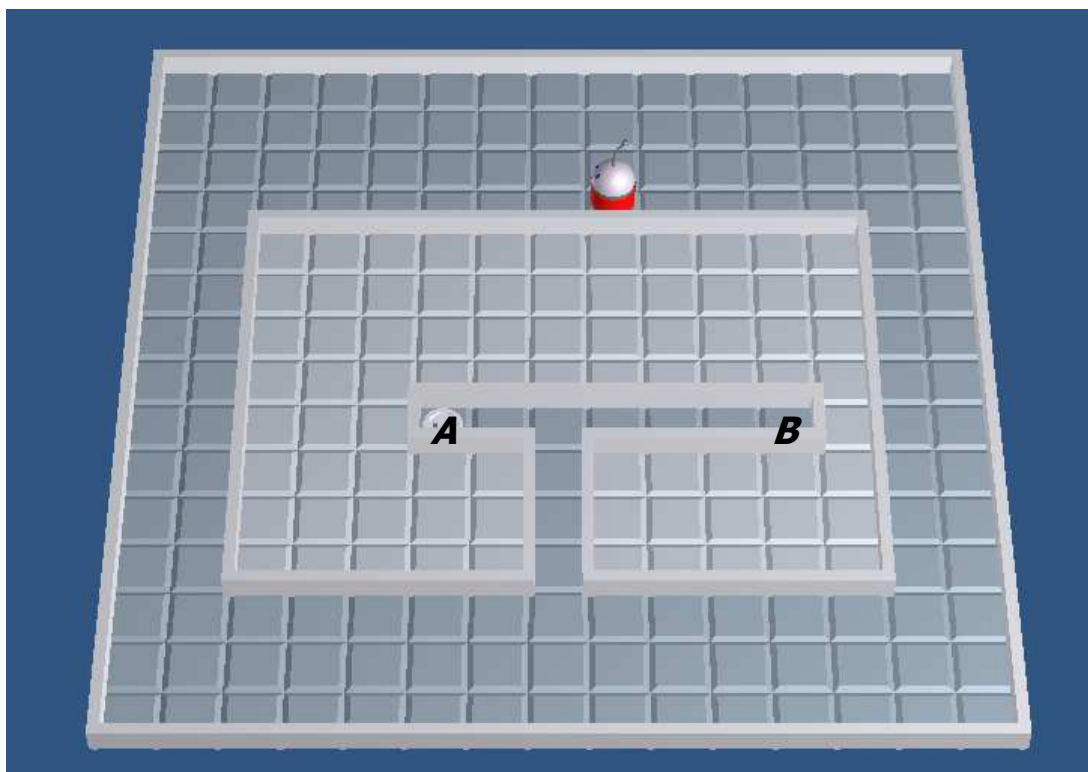
Consignes particulières :

- Pour rédiger votre algorithme, préférez le langage pseudo-code classique au 'langage' semi-graphique du logiciel
- Privilégiez une écriture qui rende vos procédures réutilisables

- ... et une autre mission RobotProg (interro de novembre 2009)

Objectif à atteindre :

Atteindre la prise de courant, s'y recharger et s'arrêter



Conditions initiales :

<i>Terrain :</i>	une construction en murs droits perpendiculaires à l'intérieur du terrain une seule ouverture débouchant sur trois couloirs perpendiculaires 'en T' et de longueurs respectives inconnues au fond d'un des deux couloirs (en A ou en B), une prise
<i>Position du robot :</i>	n'importe où le long du mur extérieur, comme illustré
<i>Direction du robot :</i>	sens anti-horlogique, comme illustré

Éléments du langage autorisés :

<i>Procédures :</i>	avancer, tournerGauche, tournerDroite, recharger
<i>Fonctions logiques :</i>	murEnFace, murAGauche, murADroite, robotSurUnePrise
<i>Fonctions numériques :</i>	aucune
<i>Instructions :</i>	tantQue, Si Alors Sinon, procédure

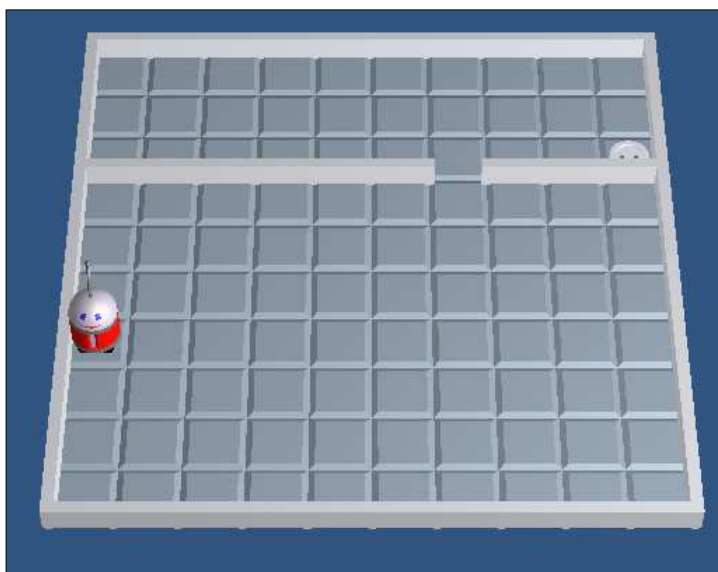
Consignes particulières :

- Pour rédiger votre algorithme, préférez un langage pseudo-code classique au 'langage' semi-graphique du logiciel
- Privilégiez une écriture qui rende vos procédures réutilisables et qui mette en évidence les états-buts du problème

- ... et encore une mission RobotProg

Objectif à atteindre :

Passer dans l'autre pièce, atteindre la prise, s'y recharger et s'arrêter



Conditions initiales :

<i>Terrain :</i>	un mur horizontal percé d'une porte (position indéterminée)
<i>Position du robot :</i>	n'importe où dans la 'pièce du bas' mais le long d'un mur (y.c. un coin)
<i>Position de la prise</i>	n'importe où mais contre un mur (y.c un coin) dans la 'pièce du haut'
<i>Direction du robot :</i>	comme vous voulez

Éléments du langage autorisés :

<i>Procédures :</i>	avancer, tournerGauche, tournerDroite, marquerCase, recharger
<i>États logiques :</i>	murEnFace, murAGauche, murADroite, caseMarquée, RobotSurUne-Prise
<i>Langage algorithmique :</i>	TantQue, Si Alors Sinon, Procédure

Consignes particulières :

- Pour rédiger votre algorithme, préférez le langage pseudo-code classique au 'langage' semi-graphique du logiciel
- Privilégiez une écriture qui rende vos procédures réutilisables

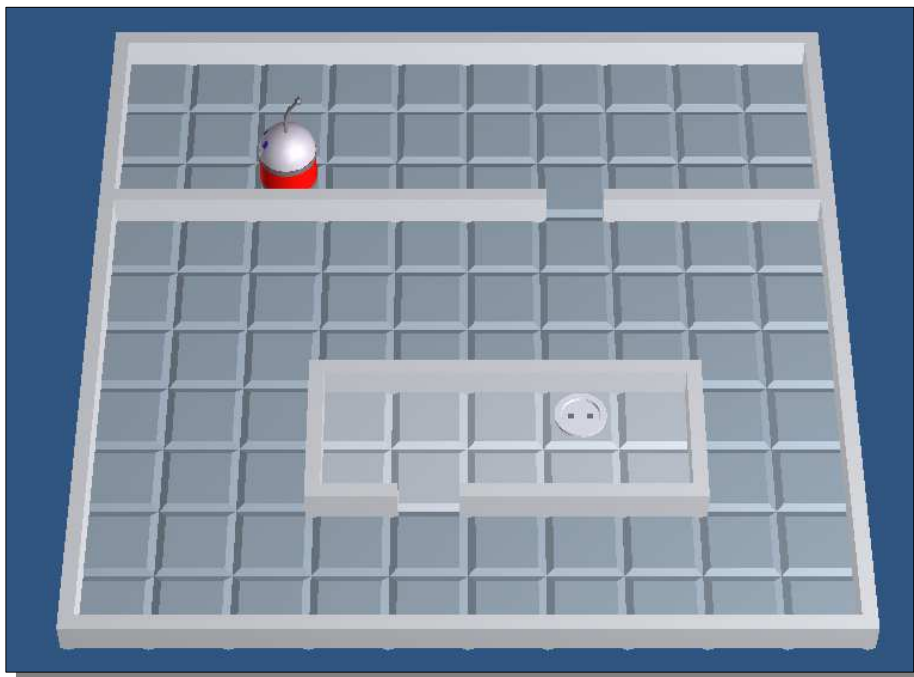
Conseils :

- Décomposition en sous-problèmes (chacun possédant son propre but) vivement conseillée

- ... et encore une mission RobotProg (interro de novembre 2010)

Objectif à atteindre :

Atteindre la prise, s'y recharger et stop



Conditions initiales :

Terrain :	<ul style="list-style-type: none"> - séparé en deux par un mur horizontal percé d'une porte - dans la partie inférieure, <u>face à la porte</u>, une pièce munie d'une porte et contenant une prise - la porte de la pièce a une <u>position quelconque</u> <u>mais pas</u> face à la porte dans le mur
Position de la prise	<u>quelconque</u> dans la pièce
Position du robot :	<u>quelconque</u> , mais dans la partie supérieure <u>et</u> le long d'un mur
Direction du robot :	sens horlogique

Éléments du langage autorisés :

Procédures :	<i>avancer, tournerGauche, tournerDroite</i>
Fonctions logiques :	<i>murEnFace, murAGauche, murADroite, robotSurUnePrise</i>
Instructions :	<i>tantQue, Si Alors Sinon, procédure</i>

Consignes :

- Pour rédiger votre algorithme, utilisez un langage pseudo-code classique et pas le 'langage' semi-graphique du logiciel; respectez la syntaxe de ce pseudo-code
- Privilégiez une écriture procédurale, pensez 'buts et états'

7. VALEURS, ÉTATS ET CONDITIONS ...

7.1. NOTION DE VALEUR

Nous avons défini qu'un état était une information relative au statut d'un automate (ce qu'il pouvait 'sentir' ou 'percevoir' de son environnement) ; jusqu'à présent nous n'avons utilisé que des états logiques (binaires), dont les seules valeurs possibles étaient vrai/faux (oui/non), ce qui était bien pratique puisqu'ils étaient utilisables directement comme condition dans un Si ... ou dans un TantQue ...

Mais il existe d'autres genres d'états, présentant plus de deux valeurs ... revenons à RobotProg un instant ... si l'on consulte *Menu > Aide > Résumé du langage robot*, on découvre ceci³⁷

DistanceMur	Fonction retournant le nombre de cases entre le robot et le mur le plus proche devant le robot
-------------	--

il s'agit d'un nouvel état (information) qui cette fois est un nombre entier, et qui permet au robot (et à tout moment) de connaître le nombre de cases qui le sépare du mur d'en face (la case qu'il occupe à ce moment n'étant pas comptée : c'est donc bien le nombre de cases entre lui et le mur ; quand il est contre le mur, la distance est donc 0).

Nous pourrions donc à présent écrire des versions équivalentes de la procédure AllerAuMur :

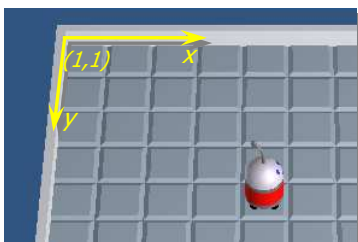
AllerAuMur : Début TantQue non MurEnFace Faire Avancer FinTQ Fin	AllerAuMur : Début TantQue non (distanceMur = 0) Faire Avancer FinTQ Fin	AllerAuMur : Début TantQue distanceMur <> 0 Faire Avancer FinTQ Fin
--	--	---

Et si l'on pousse plus avant l'examen du langage robot, on trouve aussi d'autres informations numériques entières³⁸:

xRobot	Fonction retournant la position x du robot sur le terrain
yRobot	Fonction retournant la position y du robot sur le terrain
dxRobot	Fonction retournant la direction du robot suivant l'axe x
dyRobot	Fonction retournant la direction du robot suivant l'axe y

permettant au robot (et à tout moment) de connaître sa position et sa direction, encore faut-il connaître où se situe l'origine sur terrain (quel coin ?) et quelles valeurs correspondent aux (quatre) directions possibles ... (ce que l'on pourrait désigner comme le "référentiel" du robot)

origine :



c'est (pour le spectateur) le coin supérieure gauche qui est la case (1,1) en coordonnées cartésiennes (x horizontal de gauche à droite et y vertical de haut en bas)

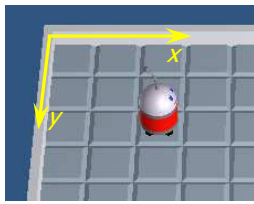
directions : remarque préliminaire : le robot ne peut se déplacer qu'en ligne droite, soit horizontalement, soit verticalement (pas de déplacement en diagonale)

- s'il n'y a pas de déplacement vertical, l'information dyRobot vaut 0
- s'il n'y a pas de déplacement horizontal, l'information dxRobot vaut 0
- si le robot s'éloigne de l'axe x, dxRobot vaut 1; s'il s'en rapproche, dxRobot vaut -1
- si le robot s'éloigne de l'axe y, dyRobot vaut 1; s'il s'en rapproche, dyRobot vaut -1

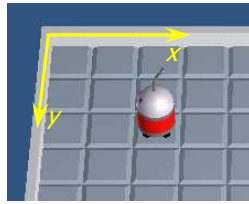
³⁷ nous reviendrons plus loin et de manière plus détaillée sur le mot **fonction** qui - s'il n'est pas utilisé ici à mauvais escient – manque cependant de définition ...

³⁸ il y a ici un terme très important que nous ignorerons provisoirement : fonction ; il fera l'objet d'une étude approfondie ultérieure

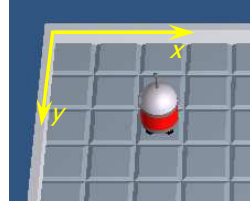
exemples ci-dessous : états du robot après avoir fait un pas (Avancer) dans la direction illustrée ...



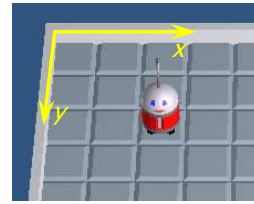
dyRobot=0 ; dxRobot=+1



dyRobot=0 ; dxRobot=-1



dxRobot=0 ; dyRobot=-1



dxRobot=0 ; dyRobot=+1

7.2. CONDITIONS SUR LES VALEURS : OPÉRATEURS DE COMPARAISON ET ÉTATS

Reprenons les versions de <AllerAuMur> :

celle de gauche utilise un état logique (valeur : vrai/faux), celles de droite une information sous forme de nombre entier (valeur : 0)

AllerAuMur :

Début

Tant que non MurEnFace Faire

Avancer

FinTQ

Fin

AllerAuMur :

Début

Tant que non (distanceMur = 0) Faire

Avancer

FinTQ

Fin

AllerAuMur :

Début

Tant que distanceMur <> 0 Faire

Avancer

FinTQ

Fin

on remarquera qu'elles sont équivalentes quant à leur logique et à leur résultat, mais pas quant aux moyens : les deux de droite utilisent un nombre (entier indiquant à combien de cases du mur se trouve le robot), celle de gauche un état logique

Que peut-on faire avec des nombres ? : calculer (additions, soustractions, multiplications, divisions, etc. ...), mais plus fondamentalement ici, on peut surtout les comparer ! Les nombres possèdent par nature une relation d'ordre ... $-3 < -2 < -1 < 0 < 1 < 2 < 3$... ce qui rend légitime l'utilisation entre eux d'opérateurs de comparaison (appelés aussi opérateurs relationnels : $=$, \neq , $<$, \leq , $>$, \geq ... ou encore : égal, différent, strictement inférieur, inférieur³⁹, strictement supérieur, supérieur)

Quelle est la réponse obtenue quand on fait une comparaison ?

une valeur logique (vrai ou faux) ; ainsi $3 > 5$ est vrai ! $5 = 2$ est faux ! ...

Dès lors, il est possible d'utiliser la négation (opérateur logique) sur une comparaison : ainsi, p.ex., non $(x = y)$ équivaut à $x \neq y$.⁴⁰

Il y a donc un rapport entre un état logique (tel que proposé par le langage robot) et des comparaisons (ici sur valeurs numériques) effectuées explicitement par le programmeur :

- l'état <MurEnFace> (vrai) correspond à une situation où **<distanceMur> = 0**
- l'état <MurEnFace> (faux) correspond à une situation où **<distanceMur> \neq 0**

Sous l'abstraction d'un état logique se 'cache' une simple comparaison entre deux états numériques, c.-à-d. entre deux valeurs.

L'évaluation d'une comparaison entre valeurs peut donc jouer un rôle de condition.

Avec ces états numériques, (et les calculs et comparaisons qu'ils permettent), on va pouvoir réaliser des algorithmes plus complexes (en fait faire face à des situations pour lesquelles les états logiques fournis étaient insuffisants) : d'une certaine manière – comme pour les actions élémentaires avec les procédures – on va pouvoir créer de nouveaux états !

³⁹ inférieur au sens mathématique, c.-à-d. inférieur ou égal ; même réflexion avec supérieur c.-à-d. supérieur ou égal)

⁴⁰ attention à ces négations de comparaisons 'non =' est \neq , 'non <' est \geq , 'non \leq ' est $>$, etc. ...

Exemples : deux 'petites' missions RobotProg :

- « s'arrêter à deux cases du mur d'en face si possible, sinon rester au point de départ ».

```

Début
  TantQue distanceMur > 2 Faire
    Avancer
  FinTQ
Fin

```

toujours le même principe : "tant que le but n'est pas atteint ..."

ici le but est une condition définie par le 'programmeur'

attention : TantQue distanceMur < 2 **et pas** TantQue distanceMur <= 2 **pourquoi ?**

- « retourner à l'origine du terrain, c.-à-d. la case (1,1) »

```

Début
  TantQue dyRobot <= -1 Faire
    TournerDroite
  FinTQ
  AllerAuMur
  TournerGauche
  AllerAuMur
Fin

```

☺ à mon tour de dire 'bof' ! vous n'aviez pas dit qu'il fallait s'exprimer en termes de but final à atteindre ?

☹ vous avez mille fois raison ! et ce serait quoi à votre avis ?

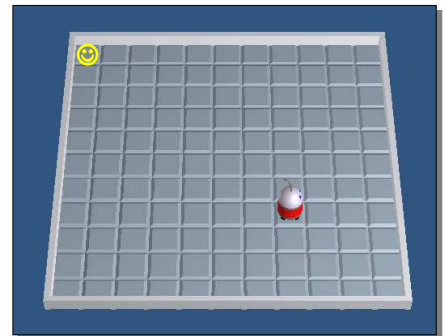
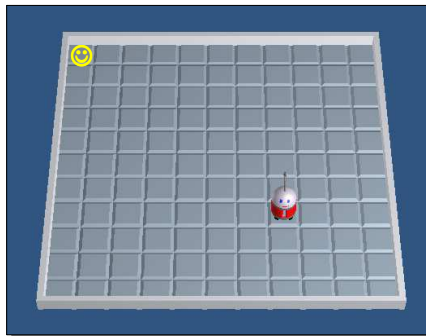
☺ TantQue non (xRobot=1 ET yRobot=1) Faire ...

☹ bravo ! ... et bien continuez, maintenant ... !

☺ ... !!! ok, mais quelles sont les conditions initiales ?

☹ comme d'habitude ... le robot peut être n'importe où (en position et en direction) ...

☺ donc le but c'est le smiley (1,1) et j'ai à ma disposition xRobot et yRobot (coordonnées courantes) pour contrôler mon déplacement, mais comment savoir dans quel sens aller ? (ci-contre p.ex., d'une manière ou d'une autre, le robot 'tourne le dos' à son objectif)



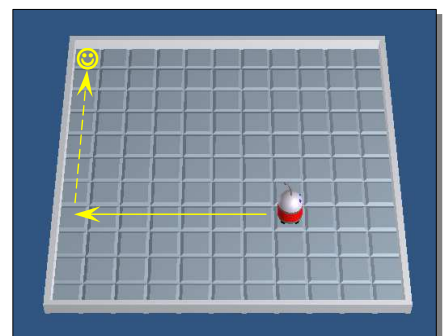
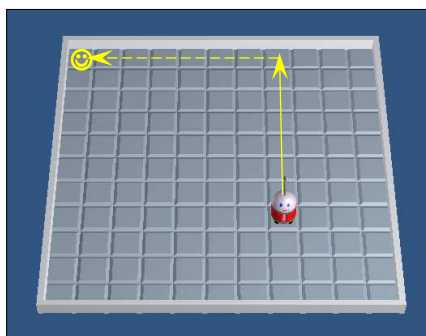
☹ n'y a-t-il pas des indicateurs de sens (ou de direction) pour vous aider ?

☺ très juste ... dxRobot et dyRobot (ici à gauche dxRobot = 0 et dyRobot = +1 ; à droite dxRobot = +1 et dyRobot = 0) ... je présume que le fait que l'objectif et le robot sont sur une diagonale est un pur hasard ?

☹ bien entendu, et il vaut mieux ne pas essayer de se déplacer autrement qu'en ligne droite ...

☺ ouiiii ... la ligne droite ! j'ai donc deux possibilités : soit atteindre d'abord l'axe x, (p.ex. à gauche) et ensuite longer cet axe dans le bon sens, soit d'abord l'axe y (à droite) et le longer aussi dans le bon sens

allez, je choisis l'axe x, et je commence par essayer de le 'viser' ; mais je dois tourner combien de fois pour être dans le bon sens ? 0, 1, 2 ou 3 fois selon l'orientation de départ



☹ *faites-en un (sous)but ! vous avez les outils ...*

☺ TantQue (dyRobot <> -1) Faire tournerDroite FinTQ ???

☹ *à votre avis ?*

☺ oui ... et j'enchaîne avec AtteindreMur TournerGauche AtteindreMur ... gagné !

☹ *oui ... mais bof ... parce que je ne vois plus aucune trace du but principal TantQue non (xRobot=1 ET yRobot=1) !!*

☺ *pfrrr* TantQue (dyRobot <> -1) Faire
 tournerDroite
 FinTQ
 TantQue non (xRobot = 1 ET yRobot = 1) Faire
 Si MurEnFace Alors
 tournerGauche
 FinSi
 Avancer
 FinTQ

☹ *et bien entendu, si le robot est au départ en (1,1), vous lui faites faire la toupie sur place, rien que pour le plaisir !*

☺ *grrrrr* TantQue non (xRobot = 1 ET yRobot = 1) Faire # but principal
 TantQue (dyRobot <> -1) Faire # sous-but
 tournerDroite
 FinTQ
 Si MurEnFace Alors tournerGauche FinSi
 Avancer
 FinTQ

☹ *et maintenant que cela marche, vous me faites le plaisir de structurer tout cela autrement, et de manière procédurale !! en utilisant p.ex.*

ViserAxeX

atteindreAxeX (qui utilisera obligatoirement soit distanceMur, soit yRobot, mais pas MurEnFace)

atteindreAxeY (qui utilisera obligatoirement soit distanceMur, soit xRobot, mais pas MurEnFace)

☺ ☹ ☹ ☹ ☹ (et j'en passe ...)

Algorithme atteindreOrigine :

Procédure viserAxeX :

Début
 TantQue (dyRobot <> -1)
 tournerDroite
 FinTQ
 Fin

Procédure atteindreAxeX :

Début
 TantQue (yRobot <> 1)
 Avancer
 FinTQ
 Fin

Procédure atteindreAxeY :

Début
 TantQue (xRobot <> 1)
 Avancer
 FinTQ
 Fin

Début

 TantQue non (xRobot = 1 ET yRobot = 1) Faire
 viserAxeX
 atteindreAxeX
 tournerGauche
 atteindreAxeY
 FinTQ
 Fin.

☹ et tout cela pour me donner le plaisir de faire quelques remarques finales :

- d'abord, vous basez votre nouvel algorithme principal sur une boucle (qui devient inutile, puisqu'elle ne s'exécutera au maximum une seule fois ...) un Si... Alors ... Sinon est de loin préférable ...
- ensuite, si vous examinez vos deux procédures atteindreAxeX et atteindreAxeY, elles ne diffèrent que par la seule comparaison explicite à xRobot et yRobot ... en lieu et place, vous pouvez ne rédiger qu'une seule procédure atteindreMur si vous utilisez distanceMur (un peu comme votre idée de départ ... pas si mauvaise finalement ... ☺)

Algorithme atteindreOrigine :

Procédure viserAxeX :

Début
 TantQue (dyRobot <> -1) Faire
 tournerDroite
 FinTQ
Fin

Procédure atteindreMur

Début
 TantQue (distanceMur > 0) Faire
 Avancer
 FinTQ
Fin

Début

Si non (xRobot = 1 ET yRobot = 1) **Alors**

 viserAxeX
 atteindreMur
 tournerGauche
 atteindreMur

 FinSi

Fin.

☺ ☠ ☠ ☠ ☠ (comme je disais ...)

7.3. NOTION DE TYPE DE DONNÉES

Les pères de l'algorithmique moderne ne se sont pas contentés d'une réflexion sur la 'représentation' des raisonnements, ils ont également pris conscience de l'importance qu'il fallait accorder aux données.

Nous venons de franchir une étape importante : la découverte de variétés différentes d'états (considérés comme des informations sur un automate et son environnement) ⁴¹:

- les états logiques (valeurs vrai/faux, p.ex. MurEnFace, CaseMarquée, RobotSurUnePrise ...)
- les états numériques (valeurs entières positives, nulles ou négatives, p.ex. distanceMur, xRobot, dyRobot)

Prenons conscience que l'on ne manipule pas de la même manière ces informations :

- on peut appliquer des opérateurs arithmétiques (+, -, * ...) aux nombres, mais pas aux valeurs logiques ...
- de même on peut appliquer des opérateurs logiques (non, et, ou) aux états logiques, mais pas aux nombres ...

Cette spécialisation cohérente de l'information est appelée typage (ou type de données) ; du point de vue strictement algorithmique⁴², la définition d'un type de données ne comporte que deux aspects essentiels :

Un type de données correspond à :

- un ensemble de valeurs distinctes (appelé domaine)
- un ensemble d'opérateurs qui ont du sens sur ce domaine

Le point capital à comprendre ici est que si un ordre existe entre les valeurs,

- alors ces valeurs deviennent comparables via les opérateurs relationnels (=, ≠, <, ≤, >, ≥) ...
- ... qu'une telle comparaison possède toujours un résultat logique (vrai/faux) ...
- ... qui peut être exploité comme condition dans une alternative (Si ... Alors ... Sinon ...) ou pour contrôler une boucle (Tant Que ...)

Exemples :

le type Logique :

- son domaine de valeurs se limite aux seules deux valeurs {vrai, faux}
- ses opérateurs sont ceux de la logique 'booléenne' : NON, ET, OU, qui permettent de construire des expressions dont le résultat est de type logique (opérateurs qualifiés d'internes)
- les valeurs logiques sont comparables entre elles (on retiendra l'égalité = et la différence ≠), le résultat d'une telle comparaison est elle-même une valeur logique

⁴¹ les deux types cités ici sont loin d'être les seuls de 'algorithmique', mais ce sont ceux que l'on utilise essentiellement dans RobotProg; d'autres types de données seront abordés dans la seconde partie du cours et une étude plus approfondie interviendra dans le cours d'«Organisation et Structure des données»

⁴² quand on passe à un niveau plus proche de la machine-ordinateur, on doit faire intervenir des considérations supplémentaires : le codage de l'information et la taille de sa représentation ; certains de ces points seront abordés dans les cours de langage si nécessaire ; mais tous feront l'objet d'un examen plus attentif dans le cours d'«Organisation et Structure des données»

le type Entier :

- son domaine de valeurs est celui des entiers relatifs (entiers négatifs, nul ou positifs)⁴³
- ses opérateurs (internes) sont l'addition, la soustraction, la multiplication, la division entière (quotient) et le reste de la division entière⁴⁴
- les entiers possèdent un ordre : chaque entier n possède un prédécesseur ($n - 1$) et un successeur ($n + 1$) ; dès lors les valeurs sont comparables ($5 < 3$?, $2 \neq 0$?), une telle comparaison possède un résultat de type logique (vrai/faux)

7.4. OUTILS DE MÉMORISATION

Nous l'avons évoqué au tout début de ce texte lors de l'introduction au concept d'automate, ce dernier doit posséder un dispositif de mémorisation, ne serait-ce que pour conserver trace de ce qu'il doit faire (c'est le rôle du mécanisme pour les automates mécaniques, et de l'algorithme pour nos automates adaptables).

Il n'y a pas que l'algorithme qui doit résider en mémoire pour pouvoir être exécuté, les états (et dans la mesure où ils peuvent posséder des types différents, nous leur donnerons désormais un nom plus générique : les données) doivent également y figurer⁴⁵... mais de quelle manière ?

a) CONSTANTES

Certaines données (valeurs), caractéristiques de l'automate ou de son environnement ne peuvent pas changer au cours de l'exécution.

De telles données sont – assez logiquement – qualifiées de constantes.

Ainsi par exemple, dans le cas de RobotProg, c'est lors de la phase antérieure de conception du terrain que les dimensions de celui-ci (nombre de lignes et de colonnes), la position des prises de courant, des murs intérieurs, etc. ... sont fixées une fois pour toutes ; leur modification en cours d'exécution de l'algorithme (de la mission à remplir) est impossible.

b) VARIABLES

D'autres données (valeurs), correspondant aux états de l'automate en cours d'exécution changent en permanence (c'est d'ailleurs le principe fondamental d'un automate adaptable : toute exécution d'une action entraîne le changement d'un ou plusieurs états, dont la détection permet la modification du comportement).

De telles données sont – tout aussi logiquement – qualifiées de variables.

Dans le cas de RobotProg, c'est à travers des noms que ces données peuvent être perçues et utilisées, p.ex. : murEnFace, xRobot, dyRobot, distanceMur⁴⁶

⁴³ contrairement aux langages informatiques, l'algorithmique ne se préoccupe pas d'éventuelles limites ou contraintes sur le domaine ; pour rester général, on pourrait exprimer cela de la manière suivante : le domaine est l'ensemble des valeurs représentables dans l'environnement de l'automate ... (le langage utilisé) ... cfr. les cours de Pascal, de C, et autres pour plus de détails ; plus de précisions dans le cours d'Organisation et Structure des Données

⁴⁴ c'est la nécessité pour les opérateurs d'être internes qui justifie qu'il faille deux opérateurs pour la division pour pouvoir exprimer : 10 divisé par 3 donne 3, reste 1


⁴⁵ John VON NEUMAN (pour rappel, auteur du premier rapport décrivant ce que devrait être un ordinateur) suggère d'incorporer les instructions de traitement dans la mémoire de l'appareil, en même temps que les données à traiter et de permettre à l'ordinateur de modifier, sur commande, ces instructions ; plus de détails dans un chapitre ultérieur ...

⁴⁶ une remarque importante à ce stade : les états logiques et numériques proposés par RobotProg ne sont des variables que pour lui-même (et non pour le programmeur) ... on en reparlera ...

c) AFFECTATION⁴⁷

L'affectation est vraisemblablement l'instruction d'action la plus importante en algorithmique impérative : elle consiste à donner une valeur particulière à une variable.

C'est l'outil fondamental de mémorisation : jusqu'à présent, nous n'avons rencontré que des situations où l'automate (ou son environnement) mettait des valeurs (états logiques ou numériques) à disposition de l'algorithme ; à présent, nous allons inverser le processus, l'algorithme va (nous allons) mettre des valeurs à disposition de l'automate.

On verra un peu plus loin qu'il existe pour cette opération des règles syntaxiques assez précises et que nous ferons usage en langage pseudo-code du symbole , comme par exemple : *variable* ← *valeur* (le récepteur - la variable - est toujours à gauche), ce qui se lit : *la variable reçoit la valeur* ou *la valeur est affectée à la variable*

Historiquement, beaucoup des premiers langages informatiques ont utilisé le même symbole = pour l'affectation (*variable = valeur*) et la comparaison (*si variable = valeur alors*)

Depuis le milieu des années 1970 s'est (quasi) généralisée une règle de bonne pratique dans les langages : un symbole différent pour chaque chose !

ainsi, en Pascal, l'affectation utilise := et la comparaison utilise =

en C (et ses descendants, java, python, ruby), l'affectation utilise = et la comparaison ==

(Malheureusement, en RobotProg c'est le même symbole = qui est utilisé pour affecter et comparer ... ☹ ... on fera avec ...)

Travaillons tout cela avec une mission RobotProg :

« *quelle que soit la position de départ – toujours le long d'un mur – faire une seule fois le tour complet du terrain, en se rechargeant au passage sur une prise (s'il y en a une)* ».

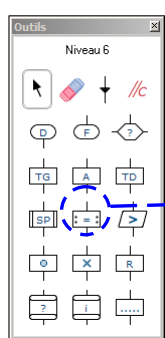
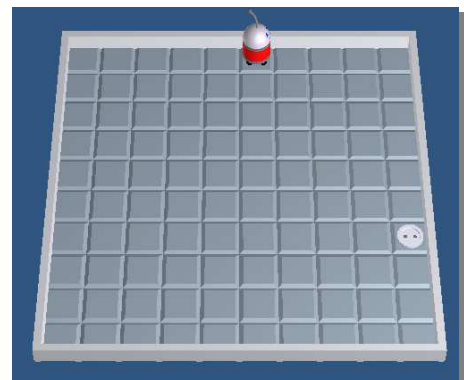
bref ... revenir (rechargé) à son point de départ !

On va donc devoir mémoriser cette position (les deux coordonnées xRobot et yRobot)

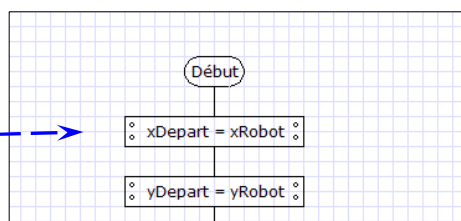
On utilisera pour ce faire deux variables (appelons-les explicitement xDepart et yDepart) auxquelles nous affecterons ces coordonnées :

xDepart ← *xRobot*

yDepart ← *yRobot*



l'affectation se trouve sur la palette (et utilise donc le symbole =)



⁴⁷ en anglais **assignment** (du verbe to assign), ce qui conduit à l'utilisation – condamnable ! – en français d'assignation au lieu d'affectation

Cette phase de l'algorithme s'appelle initialisation (mémoriser des valeurs avant de commencer), elle permet de ne pas laisser l'automate dans un état 'indéterminé' au départ ...

Une surprise nous attend à présent :

- commençons par définir le but à atteindre (l'état à rendre vrai) mais en utilisant cette fois des conditions sur les valeurs : il faut rendre vraie l'expression logique ($x_{Robot} = x_{Depart}$ ET $y_{Robot} = y_{Depart}$)

- mais si nous poursuivons de manière classique : *TantQue le but n'est pas atteint ...*

TantQue non ($x_{Robot} = x_{Depart}$ ET $y_{Robot} = y_{Depart}$) Faire ...

il est clair que le robot ne bougera pas puisque il est au départ sur la position à atteindre à l'arrivée !

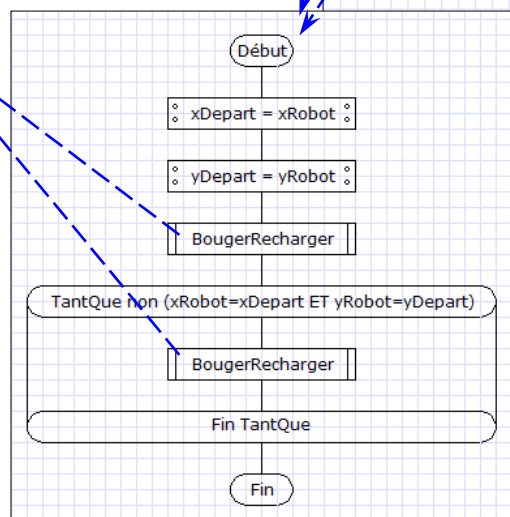
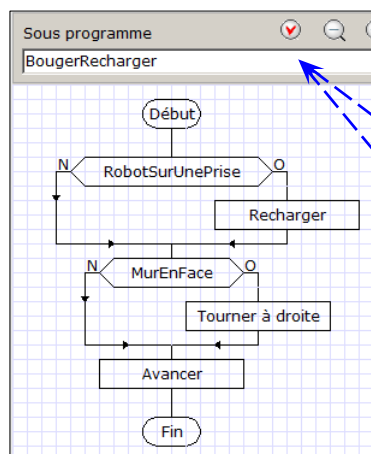
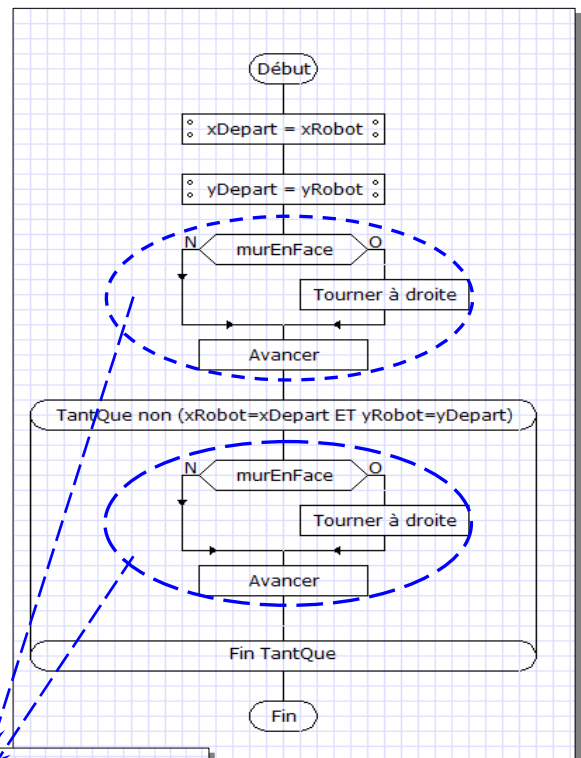
- il faut donc le faire bouger d'abord ! (*prudemment, il peut être dans le coin au départ*)

Si murEnFace Alors tournerDroite FinSi

Avancer

- ce qui donne la première version - sans se préoccuper de la prise ! – (version non procédurale, on y reviendra donc nécessairement)

on y remarque que ce que le traitement que l'on exécute dans le corps de la boucle pour avancer (*prudemment*) doit être exécuté une première fois avant même d'aborder cette boucle ; si l'on en fait une procédure (dans laquelle on traite aussi la possibilité de rencontrer une prise), cela donne :



Récrivons tout cela plus 'proprement' (on va de plus en plus se passer des ordinogrammes RobotProg et 'penser' en pseudo-code) :

Algorithme allerSimple :

```

Variable xDepart, yDepart : Entier ;    # déclaration et typage des variables utilisées
Procédure initialiserDepart :            # déclaration et définition de procédure
Debut
    xDepart ← xRobot    # mémoriser coordonnée x
    yDepart ← yRobot    # mémoriser coordonnée y
Fin
Procédure avancerRecharger :            # déclaration et définition de procédure
Debut
    Si robotSurUnePrise Alors Recharger FinSi
    Si murEnFace Alors tournerDroite FinSi
    Avancer
Fin
Debut                                    # début de l'algo (séquence principale)
    initialiserDepart                    # mémoriser point de départ
    avancerRecharger                    # quitter point de départ
    TantQue non (xRobot = xDepart ET yRobot = yDepart) Faire    # but principal : boucle
        avancerRecharger    # corps de boucle
    FinTQ
Fin

```

On rencontre tellement souvent des situations avec ce 'pattern'

action

TantQue non **condition** **Faire**

action

FinTQ

où on voit bien que le TantQue ... n'est pas adapté, parce que le principe de précaution sur lequel il est basé est devenu un obstacle à l'accomplissement du but (il empêche le démarrage, pour une fois, il faut agir avant !)

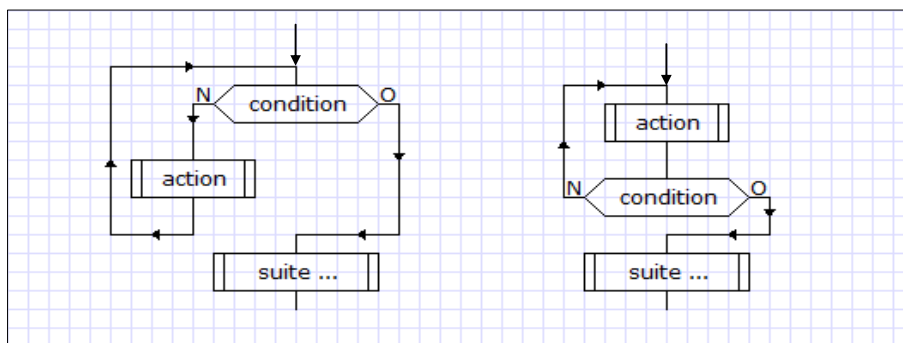
qu'on en a fait un (nouveau) type de boucle spécifique :

Repeter

action

Jusque **condition**

On reviendra plus amplement sur les différents types de boucles dans un chapitre ultérieur ; en attendant, mettons côte à côte « à l'ancienne » un TantQue ... et un Répéter ... Jusque ...



TantQue non **condition** **Faire**

action

FinTQ

suite ...

Repeter

action

Jusque **condition**

suite ...

NB : RobotProg ne possédant pas l'outil Répéter ... Jusque ..., il faudra se contenter de l'écriture équivalente⁴⁸ avec un TantQue ..., comme dans l'exemple ci-dessus

⁴⁸ c'est parce que l'écriture d'un Répéter ... Jusque ... peut se faire facilement avec un TantQue ..., alors que le contraire est beaucoup moins élégant, que l'on qualifie la boucle TantQue de primitive ...

7.5. EXERCICES

1) Objectif à atteindre (un peu compliqué) :

faire une fois le tour du terrain et s'arrêter au point de départ

Conditions initiales :

Terrain :	sans obstacle
Position du robot :	quelconque, mais 'le long' d'un mur
Direction du robot :	sens horlogique ou antihorlogique, <u>indéterminé</u>

Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite
États logiques :	--- aucun ---
États numériques :	xRobot, yRobot, dxRobot, dyRobot, distanceMur
Instructions :	tantQue, Si Alors Sinon, procédures, affectation

Consignes particulières :

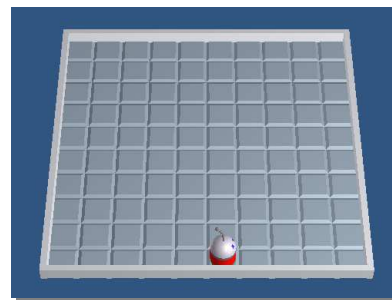
résoudre cette mission en écrivant de nouvelles procédures spécialisées et réutilisables

Indications utiles :

déterminer et mémoriser la position et l'orientation du robot au départ

dresser un tableau des combinaisons xRobot, yRobot, dxRobot et dyRobot et le synthétiser

relire les considérations de la page précédente sur les boucles



2) Objectif à atteindre (un peu plus compliqué) :

atteindre la prise en longeant le mur, s'y recharger et stop

Conditions initiales :

Terrain :	murs en 'escalier montant' prise au fond d'un couloir horizontal de largeur 1 case
Position du robot :	comme illustré !
Direction du robot :	comme illustré

Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite, recharger
États logiques :	murEnFace, murADroite, murAGauche, robotSurUnePrise
États numériques :	--- aucun ---
Instructions :	tantQue, Si Alors Sinon, procédures

Consignes particulières :

résoudre cette mission en écrivant de nouvelles procédures spécialisées et réutilisables

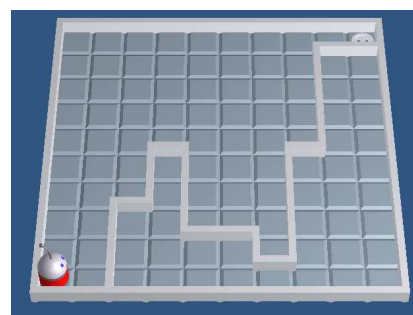
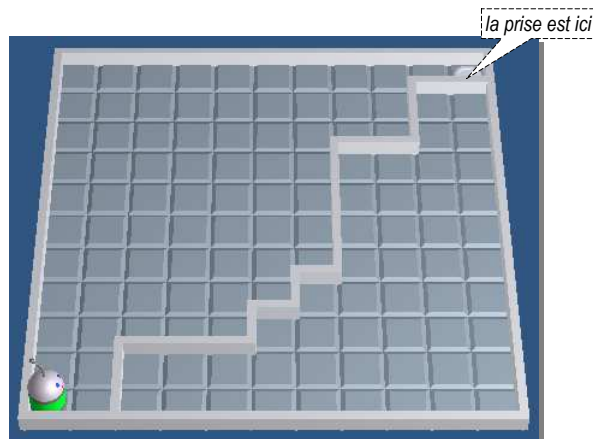
Indications utiles :

le robot part et arrive en avançant dans la même direction ...

relire les considérations de la page précédente sur les boucles

Variantes intéressantes :

- 1) modifier la taille des marches, pour voir ...
- 2) et si on ajoutait une partie 'descendante', pour voir ... ?
(on a bien dit en longeant ☺ le mur !)

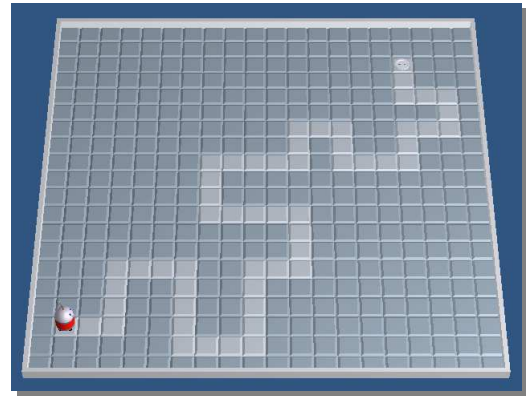


3) Objectif à atteindre (assez compliqué) :

atteindre la prise en suivant le dallage, s'y recharger et stop

Conditions initiales :

<i>Terrain :</i>	<i>dallage à angle droits ; une prise sur une dalle (pas nécessairement au bout) pas de dalle contre le mur !</i>
<i>Position du robot :</i>	<i>comme illustré !</i>
<i>Direction du robot :</i>	<i>comme illustré</i>

**Éléments du langage autorisés :**

<i>Procédures :</i>	<i>avancer, tournerGauche, tournerDroite, recharger</i>
<i>États logiques :</i>	<i>murEnFace, murADroite, murAGauche, robotSurUnePrise, sorti</i>
<i>États numériques :</i>	<i>dxRobot, dyRobot</i>
<i>Instructions :</i>	<i>tantQue, Si Alors Sinon, procédures, affectation</i>

Consignes particulières :

*résoudre cette mission en écrivant de nouvelles procédures spécialisées et réutilisables ;
pour rappel, l'état **non sorti** est vrai si le robot est sur une dalle grise
le robot ne peut faire qu'un seul pas en-dehors du dallage !*

Variantes intéressantes :

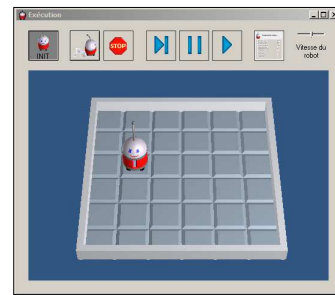
et s'il y avait des dalles contre le mur extérieur ?

7.6. INTERACTION AVEC LE « MONDE EXTÉRIEUR »

Il ne faudrait pas l'oublier, tout ce que nous disons et faisons à propos des automates programmables (et RobotProg en particulier comme outil pratique) n'est qu'un apprentissage 'en douceur (?)' des techniques algorithmiques qui seront utilisées ensuite pour programmer un ordinateur, perçu comme un 'super-automate programmable'.

Un algorithme, destiné à être traduit dans un langage de programmation particulier pour être ensuite exécuté sur un ordinateur, a besoin de pouvoir communiquer avec son environnement immédiat (disons un utilisateur) : il dispose pour cela de périphériques d'entrée/sortie, typiquement un écran (ou une imprimante) pour afficher (imprimer) des messages et/ou des résultats, et d'un clavier pour permettre à l'utilisateur d'introduire des données.

par exemple, dans RobotProg, au moment du lancement de l'exécution – via le bien nommé bouton [INIT] - l'utilisateur a la possibilité de 'placer' et d'orienter' le robot à sa guise par des clics de souris constituant dont le périphérique d'entrée; on pourrait imaginer en lieu et place que l'utilisateur communique les valeurs initiales des variables/états xRobot, yRobot, dxRobot et dyRobot depuis le clavier ...)



Si l'on veut qu'un automate dispose de ces deux capacités de communication, il est donc nécessaire de le doter de deux procédures spécialisées :

- afficher() : pour écrire des messages et/ou données sur le périphérique de sortie (écran)
- lire() : pour faire l'acquisition des données introduites par l'utilisateur sur le périphérique d'entrée (clavier) ; *qu'il n'y ait pas de confusion ici : l'automate lit dans le clavier ce que l'utilisateur y a écrit !*

la lecture est en fait une affectation d'une valeur à une variable, mais 'depuis l'extérieur' :

- $variable \leftarrow valeur$: affectation 'interne' d'une valeur à la variable (au sein de l'algorithme lui-même)
- $lire(variable)$: affectation 'externe' d'une valeur introduite par l'utilisateur depuis un périphérique d'entrée (p.ex. le clavier)

a) AFFICHAGE DE MESSAGES

Revenons encore une fois à RobotProg, sur base d'une mission précédente :

« quelle que soit la position de départ – toujours le long d'un mur – faire une seule fois le tour complet du terrain, en se rechargeant au passage sur une prise (s'il y en a une) ; le tour fini, communiquer à l'utilisateur sur combien de prises le robot est passé ».

c'est en somme assez simple, du moins à première vue; chaque fois qu'on avance, une alternative basée sur l'état <RobotSurUnePrise> permet d'augmenter d'une unité le nombre total de prises (on a donc besoin d'une telle variable) et de se recharger en énergie

il ne faut cependant pas perdre de vue :

- que cette variable doit être initialisée à 0 en début d'exécution
- qu'il faut tester l'état avant d'avancer (on est peut-être sur une prise au départ ...)



Algorithme compterLesPrises :

Variable xDepart, yDepart, **nombrePrises** : Entier # déclaration et typage des variables utilisées

Procédure initialiserDepart : # déclaration et définition de procédure

Debut

xDepart ← xRobot

mémoriser coordonnée x

yDepart ← yRobot

mémoriser coordonnée y

nombrePrises ← 0

mettre à 0 le compteur de prises

Fin

Procédure avancerCompterRecharger : # déclaration et définition de procédure

Debut

Si robotSurUnePrise Alors

nombrePrises ← **nombrePrises** + 1 # incrémenter le compteur de prises

Recharger

FinSi

Si murEnFace Alors tournerDroite FinSi

Avancer

Fin

Debut # début de l'algo (séquence principale)

initialiserDepart

Repeter

boucle répéter, pour exécuter au moins une fois le corps de boucle

avancerCompterRecharger

Jusque (xRobot = xDepart ET yRobot = yDepart)

condition d'arrêt

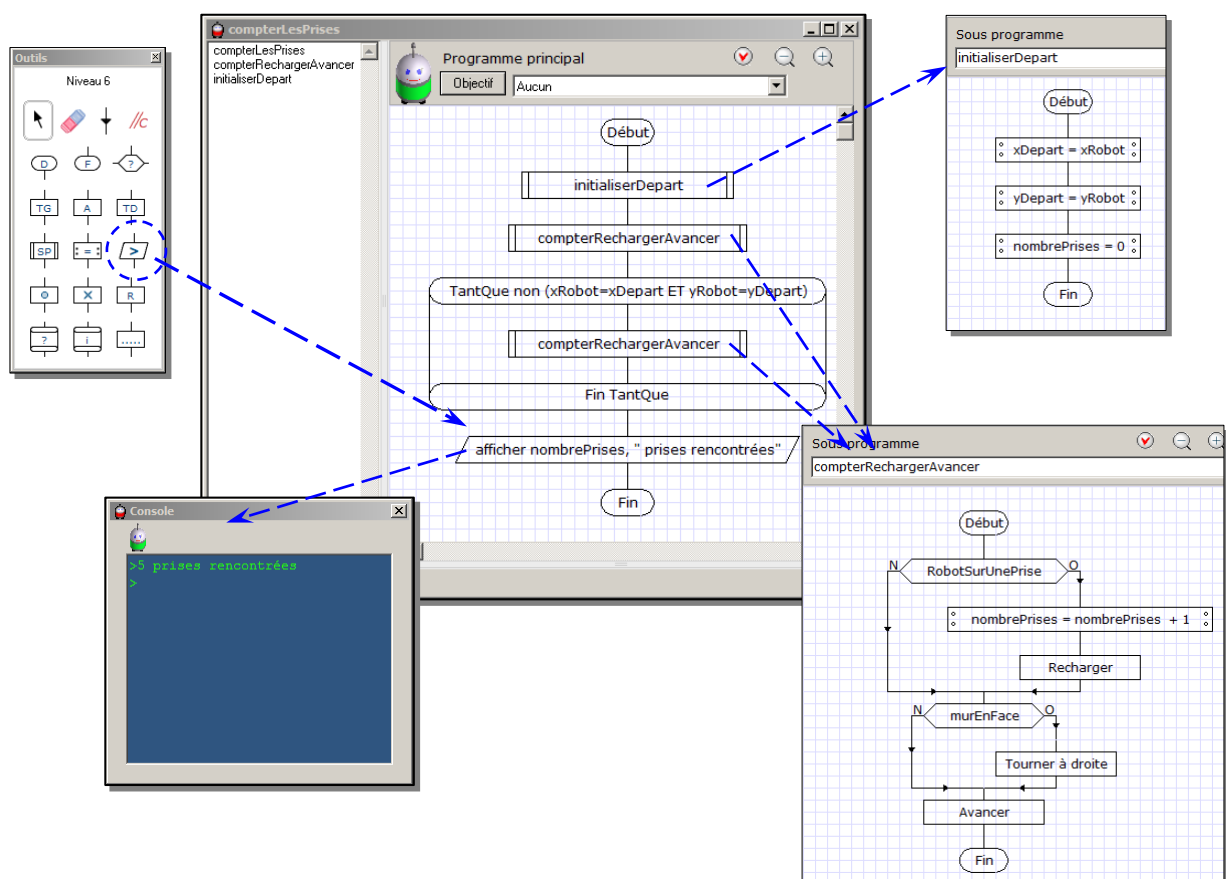
afficher(nombrePrises, 'rencontrées')

fini ! afficher le compteur de prises

Fin.

Remarque : nous avons volontairement utilisé la boucle Répéter ... Jusque ... (l'écriture équivalente au moyen du TantQue ... est laissée au soin du lecteur ... qui n'a en fait qu'à regarder ci-dessous) puisque la boucle Répéter ... Jusque n'existe pas dans RobotProg ...

cet algorithme devient (les procédures d'entrée/sortie sont symbolisées par la même icône de la palette, il suffit de commencer par le verbe 'afficher' ou 'lire') :



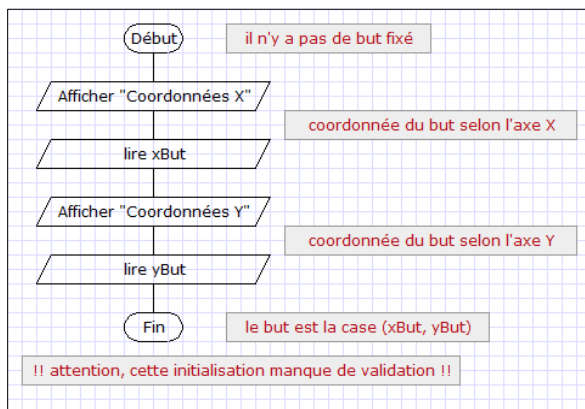
b) LECTURE DE DONNÉES

Comme annoncé précédemment, la 'lecture' n'est rien d'autre qu'une affectation de variable depuis l'environnement extérieur ; quelle que soit sa syntaxe particulière (dans un langage de programmation particulier), la procédure lire() doit spécifier le nom de la variable 'à lire' (c.-à-d. qui recevra la valeur).

Dernière (?) contribution de RobotProg à cette introduction aux « bases du raisonnement » :

« en partant de l'origine, amener le robot sur une case particulière, dont les coordonnées sont choisies par l'utilisateur ...

on écrira une procédure fixerBut qui établira un dialogue avec l'utilisateur, p.ex. comme ceci : ».



☺ c'est une variation sur 'aller à l'origine' que j'ai écrite précédemment ... je suis volontaire !

☹ ok, mais alors, profitez-en pour écrire l'algorithme directement en pseudo-code ...

☺ pourquoi pas ?

algorithme allerAuBut :

Debut

fixerBut

tantQue non (xRobot=xBut ET yRobot=yBut) Faire

????

FinTQ

Fin

parce qu'on est peut-être sur le but ...

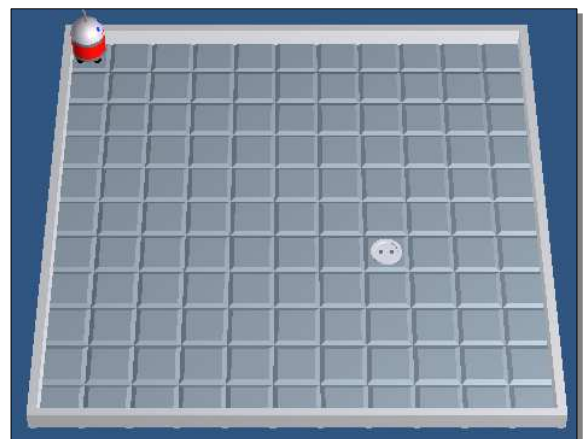
je cale un peu, là ! ce n'est pas la même chose que aller en (1,1) ; là je savais à l'avance où je devais aller : m'orienter d'abord par rapport à l'origine, puis atteindre un mur, tourner et atteindre un mur ... cette fois, je dois aller « n'importe où ! » et surtout, j'ai deux orientations possibles (tourner dans le sens des aiguilles d'une montre ou dans le sens contraire) !

☹ pas mal votre début ... et si vous considériez le but à atteindre par rapport à l'origine ?

☺ comment cela ?

☹ vous connaissez les coordonnées de votre position et les coordonnées de votre but ... et le robot connaît son orientation ...

☺ ??? ... vous voulez dire que je dois travailler avec xRobot, yRobot, xBut, yBut, dxRobot et dyRobot ... je m'en doute bien !



7.7. PAUSE ... BASES D'UN LANGAGE ALGORITHMIQUE COHÉRENT⁴⁹ (2)

À partir du moment où un algorithme va explicitement utiliser des données, la description précise de celles-ci va devoir y figurer ; un algorithme possèdera donc deux parties distinctes (et dans cet ordre !) :

- une partie déclarative : y seront spécifiés les types, les constantes et les variables nécessaires à la mémorisation des données, c.-à-d. aux différents états de l'automate-programme (c'est la partie « quoi ? »)
- une partie descriptive : rien de nouveau ici, c'est l'expression de la logique de la solution au problème posé sous forme d'une séquence d'instructions (c'est la partie « comment ? »)

a) les Types, Constantes et Variables

La notion de type a été abordée précédemment (cfr. 7.3) ; elle ne nécessite pas de commentaires supplémentaires pour l'instant ; gardons en mémoire ses caractéristiques essentielles :

- le domaine de valeurs
- les opérateurs internes
- l'ordre entre les valeurs qui permet la comparaison et donc l'expression de conditions

Les seuls types qui présentent de l'intérêt pour l'instant sont le type Logique et le type Entier (cfr. 7.3).

Comme pour la partie descriptive (instructions), les données seront déclarées en pseudo-code, donc avec une syntaxe précise :

Constantes

le bloc déclaratif des constantes commence avec le mot réservé Constante, suivi d'une liste de couples '*nom = valeur littérale*' (*le = signifie* : « la constante ayant tel nom possède telle valeur »)

Constante *nomDeLaConstante* = valeur littérale

exemples : Constante nombreCasesX = 10, nombreCasesY = 9

Variables

le bloc déclaratif des variables commence avec le mot réservé Variable, suivi d'une liste de noms (de variables) et de leur appartenance à un type précis, selon la syntaxe (*cette fois, il faut un* : pour exprimer « la variable ayant tel nom appartient à tel type »)

Variable *nomDeLaVariable* : *nomDeType*

exemples : Variable xBut, yBut : Entier # deux variables entières (pour mémoriser des coordonnées)
 butAtteint : Logique # une variable logique (pour mémoriser un état)

NB : la déclaration n'est pas une initialisation : après déclaration le contenu de la variable est indéfini

⁴⁹ les concepts du langage algorithmique pseudo-code présentés sommairement ici seront approfondis dans la seconde partie de ce texte ...

b) l'Affectation

dès lors qu'une variable a été déclarée, un contenu (une valeur) peut lui être affecté ; la syntaxe de cette action (*oui, c'est bien une action, puisqu'elle modifie l'état/valeur de la variable*) est la suivante :

nomDeLaVariable ← contenu

par contenu, on entend : toute expression dont la valeur possède le type déclaré pour la variable (le contenu peut donc être une valeur littérale, une constante, le contenu d'une autre variable de même type, toute expression construite à l'aide des opérateurs du type)

exemples :

xBut ← 6	# valeur littérale entière
xBut ← nombreCasesX	# constante entière
xBut ← yBut	# autre variable entière
xBut ← xBut + 2	# expression arithmétique entière
butAtteint ← faux	# valeur littérale logique
butAtteint ← (xBut = nombreCasesX)	# résultat de comparaison

l'affectation est **destructive** ! : elle 'écrase' l'ancien contenu de la variable par le nouveau

Note nécessaire : les noms de constantes et de variables sont appelés 'symboliques', parce que eux aussi permettent d'utiliser de l'abstraction en cachant les détails internes du 'comment ?'

On sait que, comme l'algorithme traduit (le programme compilé), les données doivent figurer en mémoire dans l'ordinateur au moment de l'exécution ; l'utilisation de noms remplace la contrainte des débuts de l'histoire de la programmation où l'on devait spécifier explicitement les adresses-mémoire ; aujourd'hui, l'association nom-adresse est prise en charge par le compilateur ; de plus, la mention d'un type de données renforce la cohérence (on délègue au compilateur la mission de valider la consistance entre le contenu de la variable et le type annoncé et de vérifier qu'on utilise bien que les seuls opérateurs permis par le type)

c) les Instructions d'Entrée/SortieÉcriture

on utilisera la procédure *écrire()* en spécifiant entre parenthèses ce que l'on souhaite afficher sur le périphérique de sortie (p.ex. écran) ; cette procédure semble 'magique' (à nouveau le 'comment ?' est caché) puisqu'on peut lui communiquer une liste d'éléments à afficher

écrire(contenu)

exemples :

écrire(6)	# valeur littérale
écrire(nombreCasesX)	# constante
écrire(xBut)	# variable
écrire(nombreCasesX - xBut)	# expression
écrire('vous êtes sur la case de coordonnées (' , xBut, ',', yBut, ')')	# texte + valeurs

Lecture

la 'lecture' n'est rien d'autre qu'une affectation 'externe' de valeur à une variable, la valeur étant celle introduite par l'utilisateur depuis le périphérique d'entrée (p.ex. clavier) ; elle obéit donc aux mêmes règles que l'affectation 'interne' décrite précédemment : cohérence entre la valeur introduite et le type de données annoncé pour la variable (sinon : résultat imprévisible ne dépendant que de la façon dont le système a été conçu : habituellement erreur fatale lors de l'exécution entraînant l'arrêt de celle-ci)

lire(nomDeLaVariable)

exemples :

lire(xBut)	# variable entière
------------	--------------------

d) la Structure d'un Algorithme

la structure générale d'un algorithme ébauchée au §6.3 comporte à présent

- une partie déclarative composée (si nécessaire, mais dans cet ordre strict !)
 - de la déclaration de l'algorithme lui-même (il doit avoir un nom)
 - de la déclaration des constantes
 - de la déclaration des variables
 - de la déclaration des procédures utilisées dans la séquence principale
- une partie descriptive composée de la séquence principale (entre les marqueurs de début et de fin)

exemple : puisque deux langages de programmation différents (le Pascal et le C) sont adossés au présent cours de 'Principes de Programmation'), nous admettrons ici deux représentations structurelles d'un algorithme qui ne diffèrent que par la place occupée par la description des procédures;

dans les deux cas, une procédure doit nécessairement avoir été déclarée et décrite avant de pouvoir l'invoquer,

- les procédures forment un bloc déclaratif placé en-dessous du bloc déclaratif des données (constantes et variables) et avant la séquence principale (ci-dessous à gauche, présentation "à la Pascal")

```

Algorithme nomAlgorithme :

Constante liste de déclarations

Variable liste de déclarations

  Procédure nomProcédure :
    Début
      séquence
    Fin

  Début
    séquence principale
  Fin
  
```

- les procédures forment un bloc déclaratif placé avant tout le reste (ci-dessous à droite, représentation "à la C")

```

Procédure nomProcédure :
  Début
    séquence
  Fin
  
```

```

Algorithme nomAlgorithme :

Constante liste de déclarations

Variable liste de déclarations

  Début
    séquence principale
  Fin
  
```

7.8. EXERCICES

- 1) Cette fois, imaginez que vous n'êtes plus l'utilisateur de RobotProg, mais le réalisateur de ce logiciel !

Vous vous apprêtez à écrire la procédure <Avancer> et vous disposez pour ce faire d'un certain nombre de constantes et de variables (présentées à la page 63), certaines de type entier, d'autres de type logique (bien entendu, vous avez la possibilité de modifier ces variables) :

constantes :

<xMax> et <yMax> : taille du terrain, dans les dimensions x et y (en nombre de cases)

variables entières

<xRobot>, <yRobot> : position courante en x et en y sur la grille

<dxRobot>, <dyRobot> : direction courante selon l'axe x et l'axe y (-1, 0, +1)

<distanceMur> : nombre de cases entre la position courante et le mur d'en face

variables logiques

<murEnFace>, <murAGauche>, <murADroite> : états

Il ne vous reste plus qu'à vous y mettre, en pensant bien à tout (il y a une certaine redondance entre toutes ces données) ! il faut que cette procédure aie strictement le comportement attendu (tout simplement d'avancer d'une case quelque que soient la position et la direction courantes, en ce compris la possibilité de 'tuer' le robot et l'affichage du message de fin de vie correspondant !)

Constante	tailleX = ..., tailleY = ...	# limites physiques du terrain (cases)
Variable :	xRobot : 0 .. tailleX, yRobot : 0 .. tailleY	# valeur de 0 à la limite physique
	dxRobot : -1 .. +1, dyRobot : -1 .. +1	# valeur de -1 à +1 (-1, 0, +1)
	distanceMur : Entier	# nul ou positif
	murEnface, murADroite, murAGauche : Logique	

Procédure Avancer :

Début

...

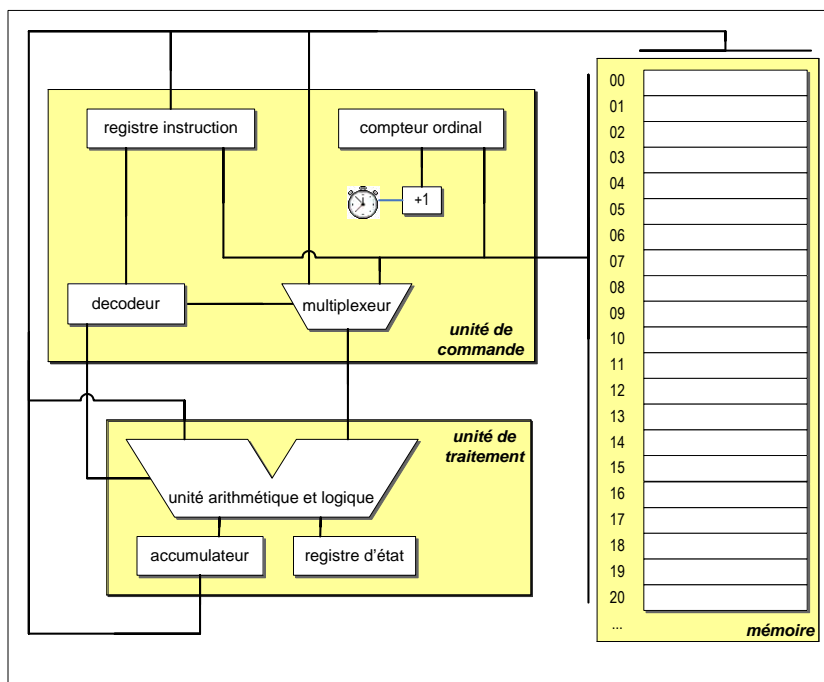
Fin.

8. L'ORDINATEUR, AUTOMATE PROGRAMMABLE ...^{50 51}

8.1. ARCHITECTURE

Il est temps de nous tourner à présent vers l'ordinateur et de découvrir comment et pourquoi il peut être considéré comme un automate programmable ...

... c'est - depuis que John von Neumann (mathématicien hongrois) en a défini l'architecture⁵² en 1945 - essentiellement un 'mécanisme' électronique correspondant au schéma (de base) ci-contre, que nous décrirons dans un premier temps en termes de composants, et dont nous expliciterons ensuite le fonctionnement.



a) COMPOSANTS

- La mémoire centrale est constitué de cases (ou cellules) numérotées (chaque case possède ainsi une 'adresse') pouvant chacune contenir une valeur (binaire, c-à-d codée à l'aide des seules valeurs élémentaires 0 et 1); cette mémoire contient le programme (les instructions) à exécuter ainsi que les données qu'il manipule et les résultats qu'il produit
- L'unité de commande (CU) et l'unité arithmétique et logique (ALU) constituent l'unité centrale (CPU⁵³), ou le processeur de l'ordinateur. Ce dernier est un assemblage de circuits électroniques qui peuvent "exécuter" des actions; aujourd'hui, il prend la forme d'un composant électronique unique nommé microprocesseur.

L'ensemble des actions ainsi "câblées" du processeur constitue son jeu d'instructions (les "actions élémentaires" dont il est capable) et détermine le langage élémentaire de son utilisation, appelé "langage machine". À chaque instruction identifiée par un code unique correspond l'activation d'un circuit particulier.

- le rôle de l'unité de commande est de recevoir et d'exécuter les unes derrière les autres - en séquence - les instructions constituant le programme; pour ce faire, elle dispose - notamment - de deux 'cases-mémoire' (appelées registres) essentielles, le registre d'instruction (IR, 'instruction register') et le compteur ordinal (IP, 'instruction pointer'), et de deux circuits jouant le rôle d'aiguillage : le décodeur et le multiplexeur
- le rôle de l'unité arithmétique et logique est d'effectuer les opérations de calcul (addition, soustraction ...) et de comparaison (permettant de déterminer des états binaires vrai/faux); pour ce faire, elle dispose également de dispositifs de mémorisation: le registre accumulateur (ACC) et le registre d'état (FLAG); l'ensemble UAL + ACC + FLAG est parfois qualifié (comme dans le dessin ci-dessus) d'unité de traitement

⁵⁰ la visite d'un site tel que <http://courses.cs.vt.edu/~csonline/index.html> est recommandée ...

⁵¹ ceci sera plus amplement développé dans le cours "Initiation à l'informatique"

⁵² en informatique l'architecture est la structure générale inhérente à un système informatique, l'organisation des différents éléments du système (logiciels et/ou matériels et/ou humains et/ou informations) et des relations entre les éléments. Cette structure fait suite à un ensemble de décisions stratégiques prises durant la conception de tout ou partie du système informatique, par l'exercice d'une discipline technique et industrielle du secteur de l'informatique appelée elle aussi l'architecture (**wikipedia**)

⁵³ les termes anglais sont 'Control Unit' ... 'Arithmetic and Logical Unit' ... 'et 'Central Processing Unit'

- Mémoire, unité de commande et unité de traitement sont reliées entre elles par des 'fils' assurant la propagation des informations sous forme de signaux électriques; techniquement, ces fils sont appelés bus
- Notons qu'il existe également une unité d'entrée/sortie, associée à une unité d'échange (non décrites ici) permettant la communication avec le 'monde extérieur' (p.ex. un clavier et un écran)

b) LANGAGE-MACHINE

Le rôle de l'unité de contrôle consiste à permettre l'exécution de l'action (l'instruction) voulue au moment voulu. Cette instruction peut concerner

- la mémoire : la consulter (la «lire») ou modifier son contenu (y «écrire»)
- l'unité arithmétique et logique (effectuer – sur le registre-accumulateur - une opération ou une comparaison)
- l'unité d'entrée-sortie (ramener le contenu de l'unité d'entrée dans l'accumulateur, afficher le contenu de l'accumulateur sur l'unité de sortie)
- voire même l'unité de commande elle-même (cfr. exemples aux pages suivantes)

De façon générale, une action élémentaire consiste donc à activer au sein du processeur les circuits permettant :

- de consulter ou modifier le contenu (l'état) de la mémoire ou du registre accumulateur (élément de mémoire incorporé à l'unité centrale),
- de déclencher une opération d'entrée-sortie (communication avec le monde extérieur via les périphériques d'interface avec l'utilisateur humain, p.ex. clavier et écran),
- ... et surtout ... de modifier la séquence des instructions formulées par le programme en commandant de «sauter» un certain nombre d'instructions sans les exécuter, ou de «revenir en arrière» pour répéter des instructions déjà déroulées (autrement dit de modifier l'ordre dans lequel le programme est «lu»)

Puisque, dans l'architecture de Turing – Von Neumann, universellement admise, le 'programme' doit résider en mémoire et que les cellules qui la composent ne savent contenir que des valeurs numériques, le plus simple est de faire correspondre à chaque action (circuit) élémentaire de l'unité de commande un code numérique

Pour notre découverte progressive de l'ordinateur-automate, nous proposerons ici un langage-machine très élémentaire, celui désigné par "instruction à une opérande"; pour ce faire, nous ferons les hypothèses simplificatrices suivantes :

- nous utiliserons la représentation décimale plutôt que binaire (pour une meilleure lisibilité ☺)⁵⁴
- les instructions sont codées numériquement et possèdent deux parties (on retrouve ici également cette dualité : *que faire ?* et *sur quoi faire ?*) :
 - un code-opération (sur deux chiffres de 00 à 99) : *que faire ?*
 - et une opérande (sur deux chiffres également, de 00 à 99) : *sur quoi faire ?* Cette opérande – appelée également argument - peut être soit une valeur explicite (littérale) soit la référence (l'adresse, le numéro) d'une case-mémoire
 - très important : on doit donc utiliser des codes différents pour distinguer sans ambiguïté les actions '*mettre 15 dans l'accumulateur*' et '*mettre le contenu de case-mémoire n°15 dans l'accumulateur*'; pour le premier cas, le code instruction se terminera par 0, pour le second il se terminera par 1
- une telle instruction (code + opérande) occupe une seule case-mémoire

⁵⁴ insistons bien : il ne s'agit ici qu'un mode de représentation destinée aux humains; en interne, l'ordinateur-automate est entièrement binaire

- pour l'instant, les instructions suivantes nous suffiront :

code	nature opérande	exemple	description
10	valeur	1009	mettre la valeur 9 dans l'accumulateur
11	adresse-mémoire	1152	mettre le contenu de la cellule-mémoire n°52 dans l'accumulateur
20	valeur	2011	ajouter la valeur 11 au contenu de l'accumulateur
21	adresse-mémoire	2153	ajouter le contenu de la cellule-mémoire n°53 à l'accumulateur
30	valeur	3003	soustraire la valeur 3 du contenu de l'accumulateur
31	adresse-mémoire	3155	soustraire le contenu de la cellule-mémoire n°55 de l'accumulateur
41	adresse-mémoire	4160	copier le contenu de l'accumulateur dans la cellule-mémoire n°60
00	valeur 00	0000	arrêter le programme

elles nous permettent par exemple de programmer l'algorithme suivant :

```

Algorithme tst000 :
Variable a, b, c : Entier
Début
  a ← 85
  b ← 6
  c ← a + b + 1
Fin

```

pour sa version en langage machine, nous supposons qu'aux trois variables entières a, b et c correspondent respectivement les cases-mémoire n° 17, 18 et 19

```

1085 # mettre la valeur 85 dans l'accumulateur (ACC ← 85)
4117 # copier le contenu de l'accumulateur dans la case n°17 (a ← ACC)
1006 # mettre la valeur 6 dans l'accumulateur (ACC ← 6)
4118 # copier le contenu de l'accumulateur dans la case n°18 (b ← ACC)
2117 # ajouter à l'accumulateur le contenu de la case n°17 (ACC ← ACC + a)
2001 # ajouter 1 à l'accumulateur (ACC ← ACC + 1)
4119 # copier le contenu de l'accumulateur dans la case n°19 (c ← ACC)
0000 # stop et fin

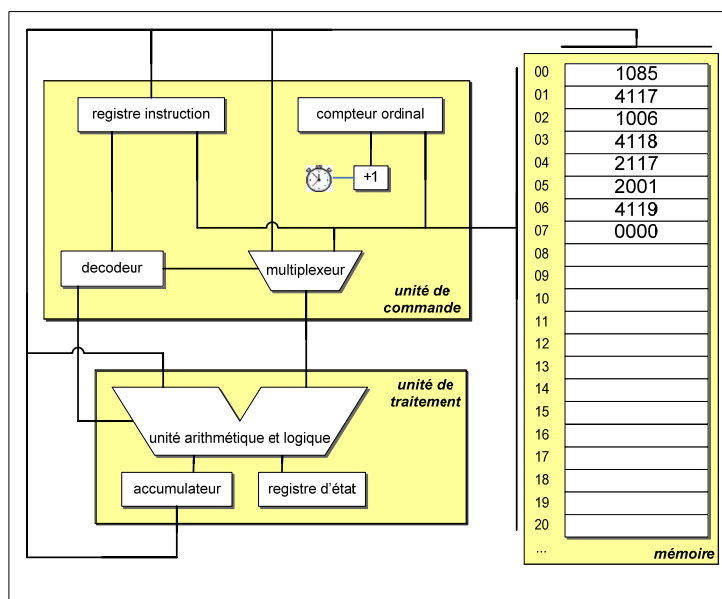
```

c) AUTOMATE STATIQUE : FONCTIONNEMENT

Commençons par placer le programme en mémoire (dans des cases successives, la première instruction occupant la case 00)

Pour pouvoir être qualifié d'automate, le dispositif ci-dessous doit – comme le lapin joueur de tambour - disposer d'un mécanisme fondé sur un cycle de base que l'on peut résumer comme suit :

- 1) prendre une instruction en mémoire et l'amener dans le registre d'instruction ...
- 2) ... faire passer le contenu du registre d'instruction dans le décodeur en séparant le code et l'opérande ...
- 3) ... faire exécuter par l'UAL l'opération correspondant au code et via le multiplexeur traiter de manière adéquate l'opérande (valeur ou adresse)
- 4) recommencer au point 1)

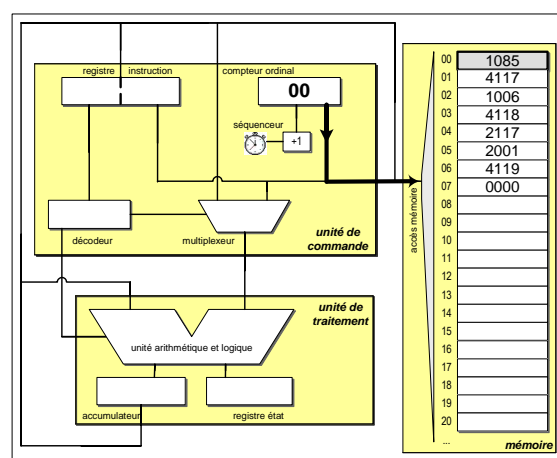
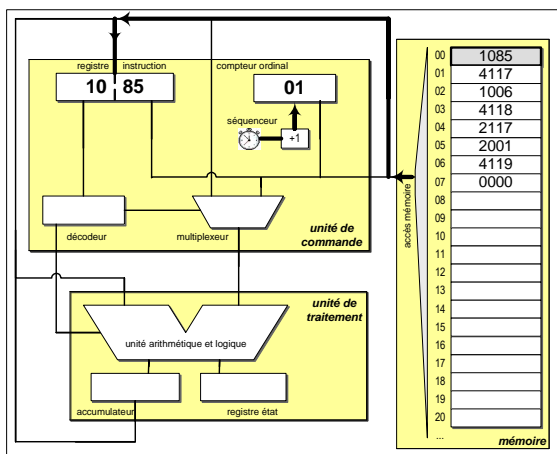


L'automatisation de ce 'cycle-machine' est assurée par le circuit séquenceur : il est organisé autour d'une horloge et du compteur ordinal dont le rôle est de désigner l'adresse de (la case-mémoire qui contient) la prochaine instruction à exécuter; une fois la première partie du cycle exécutée (transférer cette instruction dans le registre-instruction), le compteur ordinal est incrémenté, de manière à désigner l'instruction suivante à exécuter

Nous pouvons à présent analyser plus en détail le déroulement de l'exécution de notre programme : on charge donc celui-ci en mémoire et on place l'adresse de la première instruction dans le compteur ordinal :

1. prendre une instruction en mémoire et l'amener dans le registre d'instruction ...

a) l'adresse contenue dans le compteur ordinal (00) est transmise au circuit d'accès à la mémoire ... ce qui permet de sélectionner la case-mémoire correspondante

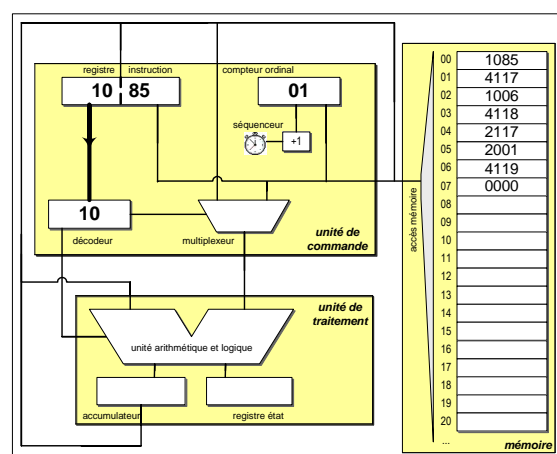
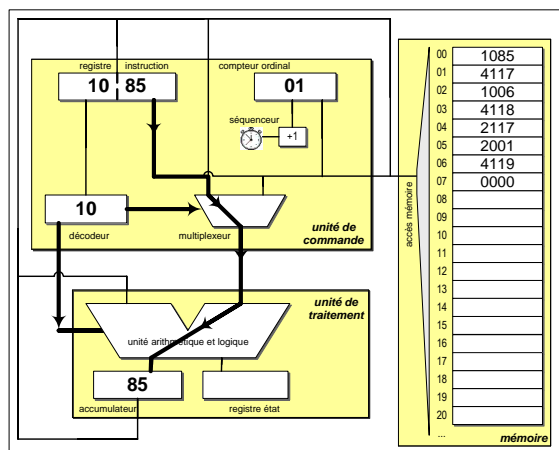


b) le contenu de cette case (l'instruction 1085) est transmise au (copiée dans le) registre-instruction et le séquenceur prépare le transfert suivant en incrémentant (+1) le compteur ordinal

2. ... faire passer le contenu du registre d'instruction dans le décodeur en séparant le code et l'opérande ...

3. ... faire exécuter par l'UAL l'opération correspondant au code et, via le multiplexeur, traiter de manière adéquate l'opérande (valeur ou adresse)

a) le décodeur ordonne à l'UAL de préparer une affectation dans l'accumulateur ...



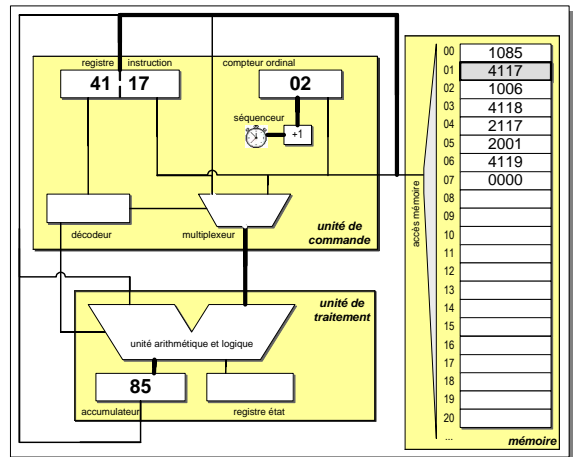
b) ... comme le code-instruction se termine par un 0 (= l'opérande est une valeur), il commande au multiplexeur de permettre à l'opérande (la valeur 85) contenue dans la partie opérande du registre instruction ...

c) ... de descendre dans l'accumulateur

fin du premier cycle ... on entame le cycle suivant ...

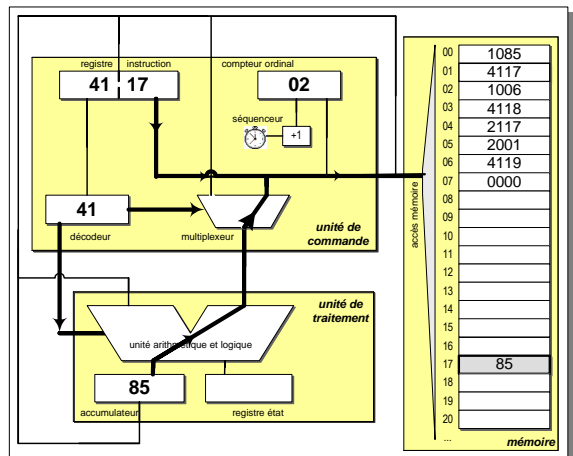
1. prendre une instruction en mémoire et l'amener dans le registre d'instruction ... (le dessin ci-contre est une synthèse, le détail et les flèches sont pour le lecteur ...)

- l'adresse contenue dans le compteur ordinal (01) est transmise au circuit d'accès à la mémoire ... ce qui permet de sélectionner la case-mémoire correspondante
- le contenu de cette case (l'instruction 4117) est transmise au (copiée dans le) registre-instruction et le séquenceur prépare le transfert suivant en incrémentant (+1) le compteur ordinal



- ... faire passer le contenu du registre d'instruction dans le décodeur en séparant le code et l'opérande ... (non illustré)
- ... faire exécuter par l'UAL l'opération correspondant au code et via le multiplexeur traiter de manière adéquate l'opérande (valeur ou adresse)

- le décodeur ordonne à l'UAL de préparer une affectation vers la mémoire
- comme le code-instruction se termine par un 1 (= l'opérande est une adresse), il commande ...
- ... au multiplexeur d'acheminer l'opérande (l'adresse 17) contenue dans registre instruction vers le circuit d'accès à la mémoire afin de permettre d'accéder à la case correspondante en écriture ...
- ... puis à l'UAL d'acheminer le contenu de l'accumulateur vers cette case

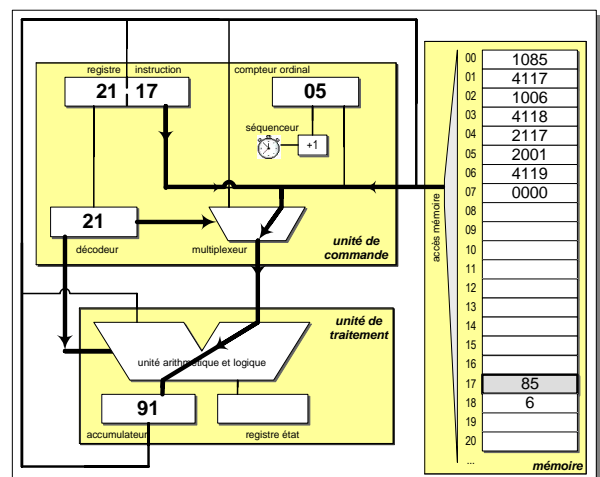


fin du deuxième cycle ... on entame le cycle suivant ... *mais, comme les deux cycles suivants (instructions 1006 et 4118) sont identiques aux deux premiers, ils sont laissés au lecteur comme exercice ...*

ce qui nous importe, c'est qu'au terme de ces deux instructions, l'accumulateur contient la valeur 6, qui vient d'être affectée à la case-mémoire 18 (correspondant à la variable b); puisque l'opération $a + b$ peut aussi s'écrire $b + a$, on va donc ajouter a au contenu de l'accumulateur (ce qui correspond à l'instruction 2117 : ajouter le contenu de la case 17 à l'accumulateur)

on connaît à présent le principe du cycle :

- le compteur ordinal contient 04, l'instruction à cette adresse est transférée dans le registre-instruction et le compteur ordinal est incrémenté
- le code-instruction est transmis au décodeur ...
- puisque le code se termine par 1, il s'agit d'un transfert-mémoire ...
- ... dans un premier temps, l'opérande (17) est transmise au circuit d'accès à la mémoire, ce qui permet d'accéder à la case correspondante en lecture
- ... puis le contenu de cette case est transmis à l'accumulateur à travers l'UAL placée en 'mode addition'



au terme du cycle de cette instruction, l'accumulateur contient donc la valeur 91

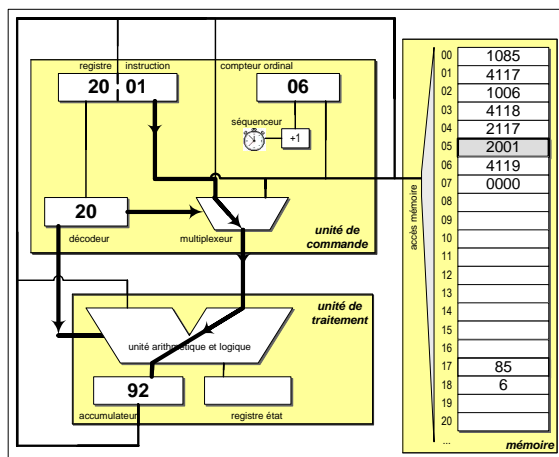
cycle et instruction suivants :

il s'agit à nouveau d'une addition, mais cette fois avec une valeur littérale (code 20) :

- le décodeur va placer l'UAL en mode 'addition'
- ...
- ... et le multiplexeur de manière à transférer l'opérande à l'UAL

l'accumulateur contient donc – enfin – le résultat de l'expression $a + b + 1$:

il ne reste plus qu'à enregistrer cette valeur dans la case-mémoire appropriée (19); *comme une telle affectation a déjà été rencontrée (instruction de code 41), elle est laissée au lecteur à titre d'exercice*



d) AUTOMATE ... ADAPTABLE ?

Avec ce qui précède, nous avons illustré des principes de base de l'ordinateur-automate, du moins la partie 'statique, figée' (action, instruction et séquence, *cfr. chapitre 5 page 5*) :

- le programme et les données sont codés de la même manière (nombres) et résident côte à côte en mémoire
- le cycle-machine est assuré par un séquenceur via le compteur ordinal
- les actions élémentaires correspondent à des affectations de valeurs (dans l'accumulateur ou dans la mémoire) ou des opérations arithmétiques (dans l'accumulateur uniquement)

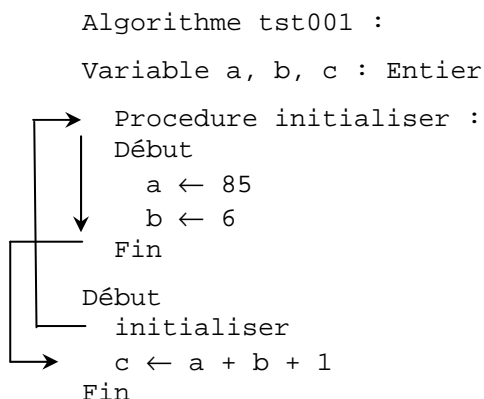
Nous allons nous pencher à présent sur la partie décisionnelle de l'automate (*cfr. chapitre 6, page 23*), ce qui permettra d'y implémenter :

- d'une part des alternatives (*Si ... Alors ... Sinon ...*)
- d'autre part des boucles (*TantQue ... Faire ...*)

... mais auparavant, il nous faut présenter une nouvelle instruction fondamentale : le branchement inconditionnel : elle 'force' le compteur ordinal en lui imposant l'adresse de la prochaine instruction à exécuter

code	nature opérande	exemple	description
51	adresse-mémoire	5112	mettre l'adresse 12 dans le compteur ordinal

rien qu'avec cette instruction, on pourrait déjà implémenter une forme rudimentaire de procédure; revisitons en ce sens notre petit algorithme en le dotant d'une procédure d'initialisation (affectation des valeurs dans les variables a et b) :



on découvre ci-contre une caractéristique importante d'un code procédural : dans un algorithme, le fait d'appeler (le terme technique est : d'invoquer) une procédure passe le contrôle aux instructions de celle-ci, qui sont exécutées en séquence de la première à la dernière; après quoi le cours 'normal' de l'algorithme continue

cette possibilité correspond à l'existence d'une instruction-machine de type 'branchement' que nous allons illustrer ...

pour sa version en langage machine, nous supposons à nouveau qu'aux trois variables entières a, b et c correspondent respectivement les cases-mémoire n° 17, 18 et 19 et que le programme est chargé en mémoire à partir de l'adresse 00

00	5106	# mettre l'adresse 06 dans le compteur ordinal
01	4117	# mettre dans l'accumulateur le contenu de la case n°17 (a)
02	2118	# ajouter à l'accumulateur le contenu de la case n°18 (b)
03	2001	# ajouter 1 à l'accumulateur
04	4119	# copier le contenu de l'accumulateur dans la case n°19 (c)
05	0000	# stop et fin
06	1085	# mettre la valeur 85 dans l'accumulateur
07	4117	# copier le contenu de l'accumulateur dans la case n°17 (a)
08	1006	# mettre la valeur 6 dans l'accumulateur
09	4118	# copier le contenu de l'accumulateur dans la case n°18 (b)
10	5101	# mettre l'adresse 01 dans le compteur ordinal

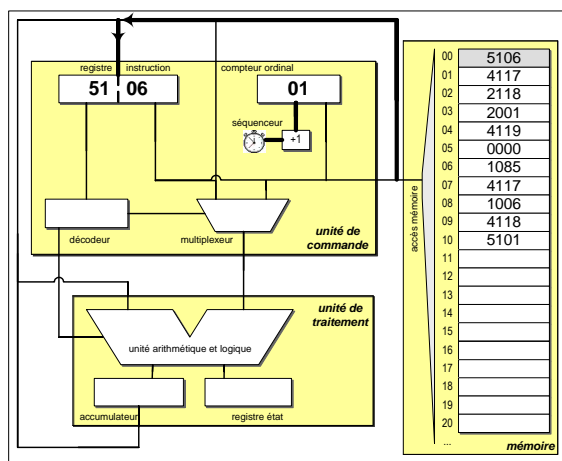
au vu de ceci, on peut comprendre qu'exécuter une procédure consisterait tout simplement⁵⁵ à exécuter une séquence d'instructions dont la première n'est pas celle prévue par le cycle normal ...

on remarque également que la fin de cette procédure comporte également un branchement vers le déroulement 'normal' du programme

comment cela se passe-t-il concrètement ? il suffit de commencer l'exécution de notre programme ...

1. prendre la première instruction en mémoire et l'amener dans le registre d'instruction ...

- a) l'adresse contenue dans le compteur ordinal (00) est transmise au circuit d'accès à la mémoire ... ce qui permet de sélectionner la case-mémoire correspondante
- b) le contenu de cette case (l'instruction 5106) est transmise au (copiée dans le) registre-instruction et le séquenceur prépare le transfert suivant en incrémentant (+1) le compteur ordinal



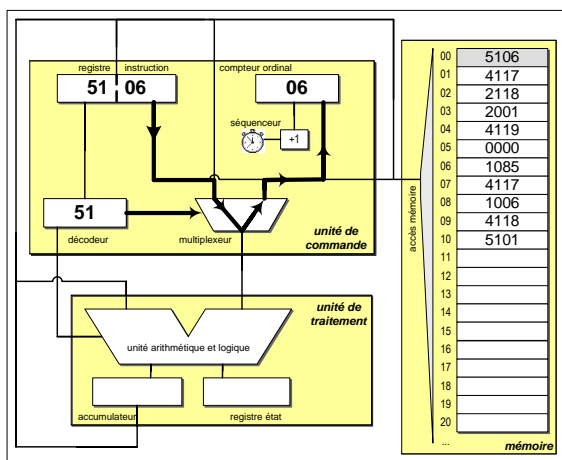
2. ... faire passer le contenu du registre d'instruction dans le décodeur en séparant le code et l'opérande ...

3. ... exécuter l'instruction

- a) ... cette fois le décodeur ne s'adresse qu'au multiplexeur, de manière à ce que ...
- b) ... l'opérande (une adresse !) se retrouve dans le compteur ordinal ...

le prochain cycle chargera donc l'instruction contenue dans la case-mémoire n°6 (1085), c'est à dire la première instruction de la procédure

l'exécution de celle-ci se déroulera en séquence ensuite, un branchement similaire (5101) assurera la reprise du cours 'normal' de l'exécution



l'exécution pas à pas de l'ensemble de ce programme est laissée au lecteur à titre d'exercice

⁵⁵ à nouveau, ceci n'est qu'une description de principe; dans la réalité, les choses sont un tout petit peu ☺ plus complexes

Avec le branchement inconditionnel (que l'on pourrait résumer par 'aller à l'adresse ...'), on imagine que l'on détient une partie importante de l'instruction de rupture de séquence qui va nous permettre d'écrire :

```
Si <condition> Alors
    aller à l'adresse ...
Sinon
    aller à l'adresse ...
FinSi
```

il nous reste donc à comprendre comment implémenter la partie Si <condition>, celle qui repose sur la détermination d'un état logique (vrai/faux) et dont on sait déjà qu'elle est basée sur une comparaison (de valeurs)

Puisque le langage-machine de notre ordinateur est très rudimentaire (pour rappel : instructions à une opérande) nos possibilités seront assez réduites ...

... une nouvelle instruction va nous permettre de comparer le contenu de l'accumulateur avec soit une valeur, soit le contenu d'une case-mémoire

code	nature opérande	exemple	description
60	valeur	6038	comparer l'égalité du contenu de l'accumulateur et de la valeur 38
61	adresse-mémoire	6017	comparer l'égalité du contenu de l'accumulateur et du contenu de la case-mémoire d'adresse 17

- s'il y a égalité (Alors), le décodeur provoque l'exécution de l'instruction suivante (le comportement normal donc, puisque le compteur ordinal a déjà été incrémenté lors du chargement de l'instruction de comparaison)
- s'il n'y a pas égalité (Sinon), le décodeur fait en sorte de 'sauter par-dessus' l'instruction suivante (en incrémentant une fois de plus le compteur ordinal)

Mais comment déterminer cette égalité ? Grâce à la partie logique de l'UAL qui comporte des circuits comparateurs : on peut considérer ici que le registre d'état va indiquer (sous forme d'état logique 0=faux, 1=vrai) le résultat de la comparaison d'égalité et que ce résultat est utilisé par le décodeur pour déterminer quelle instruction sera exécutée ensuite

exemple 1 :

Algorithme tst002 :

Variable a, b : Entier

Début

a ← 8

b ← 0

Si a = 2 Alors

b ← a

FinSi

b ← b + 1

Fin

*l'adresse-mémoire 18 correspond à a
l'adresse-mémoire 19 correspond à b
le programme en langage-machine réside en mémoire à partir de l'adresse 00 (ci-contre)*

00	1008	# ACC ← 8
01	4118	# a ← ACC
02	1000	# ACC ← 0
03	4119	# b ← ACC
04	1118	# ACC ← a
05	6002	# ACC = 2 ? (60 = comparer valeur)
06	5111	# oui : aller en 11
07	1119	# non : ACC ← b
08	2001	# ACC ← ACC + 1
09	4119	# b ← ACC
10	0000	# stop et fin
11	1118	# ACC ← a
12	4119	# b ← ACC
13	5107	# aller en 07

commentaire : dans la mesure où une instruction d'affectation telle que $b \leftarrow a$ correspond à plus d'une instruction-machine (ici, il en faut deux : charger le contenu de a dans l'accumulateur, enregistrer le contenu de l'accumulateur dans b), les instructions correspondant à Alors ... et Sinon ... doivent être des branchements (dont il faut assurer le retour !)

exemple 2 :

Algorithme tst003 :

Variable a, b, c : Entier

Début

a ← 5

b ← 6

Si a = b Alors

c ← a

Sinon

c ← b

FinSi

c ← c + 1

Fin

l'adresse-mémoire 18 correspond à a

l'adresse-mémoire 19 correspond à b

l'adresse-mémoire 20 correspond à c

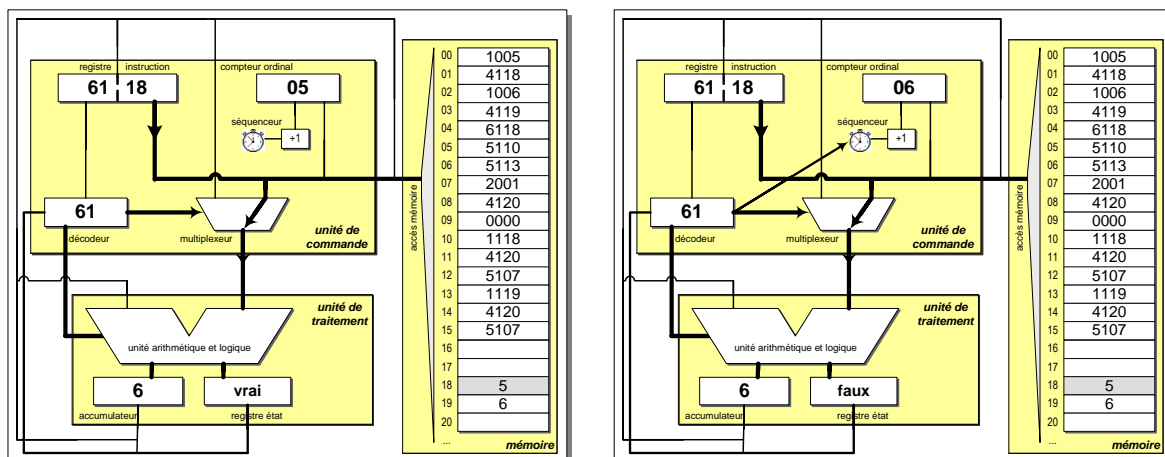
le programme commence en 00

00	1005	# ACC ← 5
01	4118	# a ← ACC
02	1006	# ACC ← 6
03	4119	# b ← ACC (ici ACC contient b, prêt pour comparaison)
04	6118	# ACC = a ? (61 = comparer contenu adresse)
05	5110	# oui : aller en 10
06	5113	# non : aller en 13
07	2001	# ACC ← ACC + 1
08	4120	# c ← ACC
09	0000	# stop et fin
10	1118	# ACC ← a
11	4120	# c ← ACC
12	5107	# aller en 07
13	1119	# ACC ← b
14	4120	# c ← ACC
15	5107	# aller en 07

commentaire : on perçoit bien ici la difficulté et la lourdeur d'écriture de programmes en langage 'bas niveau' (et ici on est au plus bas, puisqu'il s'agit du langage-machine) : toute prise de décision entraîne un 'déroutement avec retour' qu'il s'agit de rédiger soigneusement (p.ex. quel seraient les conséquences de l'oubli de la ligne 12 ?) ... nous avons eu l'occasion précédemment d'évoquer les 'langages-spaghetti' (les flèches associées au programme ci-dessus évoquent les ordinogrammes)

Côté interne, on peut imaginer que les choses se déroulent comme suit (il s'agit ici d'exécuter la ligne 04 contenant l'instruction de comparaison) ... au départ tout se déroule de la manière habituelle (le compteur ordinal contient donc l'adresse 04):

1. prendre l'instruction en mémoire et l'amener dans le registre d'instruction : l'adresse contenue dans le compteur ordinal (04) est transmise au circuit d'accès à la mémoire ... ce qui permet de sélectionner la case-mémoire correspondante; le contenu de cette case (l'instruction 6118) est transmise au (copiée dans le) registre-instruction et le séquenceur prépare le transfert suivant en incrémentant (+1) le compteur ordinal qui contient donc 05
2. ... faire passer le contenu du registre d'instruction dans le décodeur en séparant le code et l'opérande ...
3. ... exécuter l'instruction
 - a) ... l'opérande est une adresse ... qui est transmise au circuit d'accès à la mémoire, afin de ramener le contenu (5) de la case correspondante (18) vers l'ALU ...
 - b) ... placée par le décodeur en mode 'comparaison d'égalité' entre cette valeur et l'accumulateur
 - c) le registre d'état indique au décodeur le résultat logique (vrai/faux) de la comparaison :



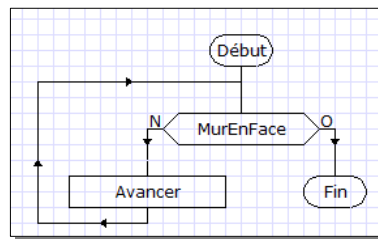
- si vrai (à gauche) : exécution normale de la prochaine instruction sur base du compteur ordinal
- si faux (à droite) : incrémentation supplémentaire du compteur ordinal, puis cycle normal ...

Et les boucles ? par exemple comment s'écrit et s'exécute un TantQue ... Faire ... ?

... tout simplement en se souvenant de ceci (rédigé il y a déjà quelques pages !)

```

AtteindreLeMur :
Début
  Tant que non MurEnFace
    Avancer
Fin
  
```



... on combine un test d'égalité (que nous savons faire faire à notre ordinateur-automate (instructions 60 ou 61), d'une séquence d'instructions (le corps de la boucle) et d'un retour au test (à l'aide d'un branchement : instruction 51)

Si l'on voulait exécuter l'algorithme suivant : faire compter l'ordinateur de 1 à 20

Algorithme tst004 :

Variable a, b : Entier

Début

b ← 20

a ← 1

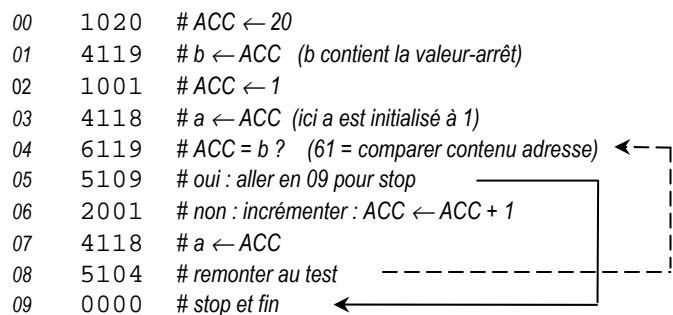
TantQue a ≠ b Faire

a ← a + 1

FinTQ

Fin

*l'adresse-mémoire 18 correspond à a
l'adresse-mémoire 19 correspond à b
le programme commence en 00*



comparer avec l'ordinogramme ci-dessus à droite

commentaire : une boucle (qu'elle soit avec test antérieur ou postérieur) n'est donc jamais qu'une comparaison sur laquelle on se 'rebranche' après exécution du corps de la boucle ...

Un petit mot sur les comparaisons par l'unité logique : se limiter à la seule égalité (et donc aussi à la différence) est évidemment insuffisant : il faut absolument couvrir tous les cas possibles (<, ≤, >, ≥, =, ≠) !

on considèrera ici que le registre d'état de l'UAL possède trois indicateurs logiques distinct, correspondant chacun à une situation "de signe" : l'accumulateur contient 0, l'accumulateur contient une valeur (strictement) négative, l'accumulateur contient une valeur (strictement) positive

de la sorte, pour réaliser la comparaison entre deux valeurs a et b, on effectue leur différence (a - b) dans l'accumulateur; si a = b, l'indicateur 'nul' est vrai, si a > b, l'indicateur '+' est vrai, si a < b, l'indicateur '-' est vrai, et la combinaison de l'indicateur 'nul' et des indicateurs de signe permettent de déterminer les autres états de comparaison

bien entendu, à tout ceci correspondent des codes-instructions appropriés ...

Voilà ... pour une rapide (!) découverte de l'ordinateur considéré comme un automate adaptable (programmable), on y a (re)découvert les 'briques' fondamentales :

- actions : instructions d'affectation de valeurs (ACC => mémoire, mémoire => ACC, ACC => ACC correspondant aux opérations numériques)
- états logiques (via registre d'état) : instructions de comparaison et de branchement permettant les conditionnelles (et donc les boucles)

8.2. LANGAGES DE PROGRAMMATION

a) DU LANGAGE-MACHINE AU LANGAGE D'ASSEMBLAGE

Même avec la machine rudimentaire que nous avons décrite précédemment (correspondant plus ou moins à des ordinateurs de 'première génération'⁵⁶ dont le langage-machine est constitué d'instructions à une seule opérande), on peut déjà mesurer la difficulté de rédiger des programmes structurés et dont la maintenance soit aisée ... si l'on dresse la liste complète des instructions strictement indispensables, elle est finalement relativement grande (une trentaine d'instructions), ce qui augmente l'effort de mémorisation⁵⁷ du programmeur et le risque d'erreurs (p.ex. confondre le code 10 et le code 11 est évidemment catastrophique)

Avec l'évolution rapide du matériel (la deuxième génération d'ordinateurs correspond à l'apparition du transistor vers 1956), on voit rapidement apparaître des processeurs présentant des jeux d'instructions

- à deux opérandes (source et destination)

p.ex. 10 05 61 # mettre la valeur 5 dans l'accumulateur et enregistrer l'accumulateur dans la case-mémoire d'adresse 61 (donc affectation de type " $a \leftarrow 5$ " en une seule ligne)

- puis à trois opérandes (source1, source2 et destination)

p.ex. 22 15 71 62 # mettre la valeur 15 dans l'accumulateur, y ajouter le contenu de la case-mémoire d'adresse 71 et enregistrer le résultat dans la case-mémoire d'adresse 62 (donc addition et affectation de type " $b \leftarrow 15 + a$ " en une seule ligne)

le gain, en termes de taille de programme, est significatif : (on fait en une seule instruction ce qui en demandait plusieurs précédemment), mais le prix à payer l'est tout autant : le nombre d'instructions est multiplié par 2 ou 3 !

par exemple, il faut quatre instructions d'addition différentes comme celle illustrée ci-dessus selon que les deux premières opérandes sont des valeurs, que la première est une valeur et la seconde une adresse, que la première est une adresse et la seconde une valeur, qu'elles sont toutes deux des adresses !

un autre bénéfice est l'apparition d'instructions de branchement conditionnel, correspondant à un (quasi vrai) si ... alors ... sinon ...,

p.ex. 81 70 18 22 # si le contenu de l'accumulateur est égal au contenu de la case-mémoire d'adresse 70, alors aller à l'instruction d'adresse 18 sinon aller à l'instruction d'adresse 22

Autre problème majeur : chaque processeur possède son propre jeu d'instructions et exécuter un programme sur des matériels différents est tout simplement impossible ! il y a une dépendance totale entre un programme et la machine pour laquelle il a été écrit.

En 1950, M.V. Wilkes 'invente' l'assembleur⁵⁸ (qui marque la fin des 'langages' de première génération et ouvre une seconde génération) qui apporte une solution à plusieurs problèmes

le principe est simple : représenter plusieurs instructions différentes (mais semblables dans leurs effets) par un même code mnémonique plus explicite et facilement mémorisable par le programmeur, et différencier la nature de l'opérande par un symbole distinct

exemple, nous avons besoin de deux instructions différentes pour charger l'accumulateur avec une valeur littérale ou le contenu d'une case-mémoire :

10 25 # mettre la valeur 25 dans l'accumulateur

11 25 # mettre le contenu de la case-mémoire d'adresse 25 dans l'accumulateur

⁵⁶ voir par exemple http://fr.wikipedia.org/wiki/Histoire_de_l'informatique

⁵⁷ bien entendu, ce que nous n'avons pas illustré dans les pages précédentes, c'est que tout est en binaire (des 1 et des 0) et non en décimal, aussi bien les codes que les valeurs et les adresses et que toute instruction prend en réalité la forme suivante 110011010101

⁵⁸ bien qu'on le dénomme également langage d'assemblage, l'assembleur n'est pas un langage de programmation mais un simple (!) convertisseur <http://fr.wikipedia.org/wiki/Assembleur> désigné également autrefois par le terme de 'langage autocode'

avec l'assembleur, le programmeur pourra désormais écrire (par exemple)

```
LOAD $25      # mettre la valeur 25 dans l'accumulateur
LOAD @25      # mettre le contenu de la case-mémoire d'adresse 25 dans l'accumulateur
```

il suffira donc d'inventer un code pour chaque type d'opération, par exemple, ...

code mnémorique	signification	effet
LOA	load	charger l'accumulateur
ADD	add	additionner
SUB	subtract	soustraire
STO	store	enregistrer l'accumulateur en mémoire
JMP	jump, branch to	branchement à ...
EQ	equal to	tester l'égalité (ACC = y)
NE	not equal to	tester l'inégalité
LT	less than	tester la stricte infériorité (ACC < y)
LE	less or equal than	tester l'infériorité (ACC ≤ y)
GT	greater than	tester la stricte supériorité (ACC > y)
GE	greater or equal than	tester la supériorité (ACC ≥ y)
...
symbole d'opérande		signification
\$		valeur littérale
@		adresse

... pour disposer d'un mode de rédaction des programmes nettement plus facile à apprendre, plus clair donc plus lisible, en particulier pour les machines à deux ou trois opérands (p.ex. ADD \$15 @63 @70)

L'assembleur est donc lui-même un programme (écrit en langage-machine, quelques dizaines à quelques centaines de lignes, tout de même) dont le rôle est simplement de générer les instructions-machine correspondant au code mnémorique et au type d'opérande, par exemple

```
LOA $25      # génère 1025
LOA @25      # génère 1125
```

On voit également apparaître un bénéfice de taille : les programmes rédigés en assembleur deviennent portables sur des matériels différents caractérisés par des codes-machine différents, il suffit que chacun possède son propre générateur d'instructions-machine (assembleur); l'indépendance programme – machine se met doucement en marche ...

Une avancée décisive vers une plus grande indépendance sera accomplie avec l'introduction de ce qu'on appellerait aujourd'hui les variables : remplacer les adresses par des noms symboliques, ce qui implique d'introduire de nouvelles instructions : les instructions déclaratives :

```
DEF A @70      # symboliser l'adresse-mémoire 70 par la lettre A
DEF B @71      # symboliser l'adresse-mémoire 71 par la lettre B
DEF C @72      # symboliser l'adresse-mémoire 72 par la lettre C
...
ADD A B C      # exécuter l'expression C ← A + B (machine à trois opérands)
```

au moment de l'assemblage (de la génération du code-machine), il suffit d'une simple substitution du symbole par l'adresse correspondante

(p.ex. en supposant que le code-instruction 24 correspond à une addition dont les deux premières opérands sont des adresses, le ADD A B C ci-dessus générera 24707172)

Avec cette étape, le programme s'affranchit des contraintes-mémoires (les adresses ne sont plus dans le code), commence à gagner en expressivité et en possibilités d'abstraction

Nous sommes au milieu des années 1950, la grande aventure des (vrais) langages de programmation dits de troisième (puis de quatrième) génération peut commencer ...

b) DU LANGAGE D'ASSEMBLAGE AUX LANGAGES DE PROGRAMMATION

Durant les années 1950, les choses vont s'accélérer, l'informatique commence à se répandre, d'abord dans le "complexe militaro-industriel" bien sûr, guerre froide oblige, dans la recherche scientifique (et nucléaire en particulier), pour le début de la conquête spatiale, et au sein de grandes entreprises – banques, assurances, ... - soucieuses d'automatiser leur gestion.

Si le coût du matériel diminue (c.-à-d. qu'à prix constant, les performances augmentent régulièrement et considérablement, ce qui sera exprimé en 1965 par la loi de Moore⁵⁹), la demande explose et le coût de développement (conception, rédaction et test de programmes) devient rapidement considérable compte tenu du caractère rudimentaire des moyens d'expression (trop proches de la machine physique, et pas assez du problème posé).

Le langage-machine dans un premier temps, et même l'assembleur ensuite, deviennent des goulots d'étranglement (on en arrive par exemple à une situation où le coût de conception et de rédaction d'un programme pour l'US Air Force est de \$72 la ligne et le coût des tests et modifications qui en découlent atteint pas loin de \$4,000 la ligne !)

Pour avancer de manière significative, il faut donc inventer de 'vrais' langages, aussi proches du langage naturel que possible (proposant un haut niveau d'abstraction, par exemple de 'vraies' structures alternatives, des boucles, des procédures ...).

Une vaste communauté (constructeurs, universités, ...) s'attelle à cette tâche : informaticiens bien sûr mais également linguistes⁶⁰ pour définir la 'grammaire' de ces "nouvelles langues" et rédiger les programmes qui en assureront la 'traduction' en langage-machine : les compilateurs. Ceux-ci sont bien plus complexes à réaliser qu'un assembleur : alors que celui-ci ne fait que de la simple conversion (1 ligne assembleur = 1 ligne de code-machine), un compilateur va devoir - à partir d'une seule ligne de programme – générer plusieurs lignes – voire plusieurs dizaines de lignes - de code-machine.

Ainsi, en quelques années (1953 -1960), on verra l'apparition

- de Fortran (formula translator), langage destiné à la communauté scientifique, très orienté 'math'
- de Cobol (common business oriented language), langage qui, comme son nom l'indique, permet de programmer des problèmes de gestion (une place toute particulière y est réservée à la persistance des données : les fichiers)
- de Lisp (list processing), langage destiné à traiter des problématiques de l'intelligence artificielle, qui émerge alors au MIT

Il faudra toutefois attendre les années 1970 - 1975 pour voir l'apparition de premiers langages réellement 'de haut niveau' (correspondant aux idées présentées et défendues dans ce cours : une approche impérative, procédurale et itérative) et en particulier ceux utilisés dans le cadre notre enseignement de première année : Pascal et C

En une cinquantaine d'années (1950 – 2000), on dit que ce sont plus de 5000 (!) langages informatiques qui ont été créés; bien entendu, certains eurent une vie très éphémère, d'autres sont toujours là (... parce qu'ils ont su évoluer); une chose est sûre, il y en avait environ 120 en 1967 et au moins 2500 en 1993; pour les curieux, un site⁶¹ les recense et les catégorise, et d'autres^{62 63} proposent une ligne du temps particulièrement instructive quant à l'évolution et aux filiations (quel langage dérive d'un autre ...).

⁵⁹ http://fr.wikipedia.org/wiki/Loi_de_Moore

⁶⁰ voir p.ex. "Hiérarchie de Chomsky" dans http://fr.wikipedia.org/wiki/Grammaire_formelle

⁶¹ <http://www.info.univ-angers.fr/~gh/hilapr/langlist/langlist.htm>

⁶² <http://www.info.univ-angers.fr/~gh/hilapr/levenez/history.html>

⁶³ <http://forge.info.univ-angers.fr/~gh/hilapr/hilapr.htm> dont la visite est recommandée ...

9. DE L'ALGORITHME ABSTRAIT AU PROGRAMME CONCRET

9.1. PRINCIPE ET EXEMPLES

Revenons à nos automates programmables⁶⁴, qu'il s'agisse de RobotProg ou p.ex. du Peleur de Patates ... dont nous avons établi l'algorithme abstrait⁶⁵ suivant (nous restons provisoirement dans l'hypothèse d'un tas de patates inépuisable) :

```

Algorithme Peleur :
Début
  TantQue non marmitePleine Faire
    Si seauVide Alors remplirSeau FinSi
    pelerPatate
  FinTQ
Fin.

```

Notre but à présent est de :

- conserver cet algorithme tel quel (idéalement, comme il est rédigé ci-dessus !) mais ...
- ... en donnant un contenu à la fois aux procédures et aux états de cet algorithme à l'aide cette fois de données concrètes (on a pu découvrir dans la partie consacrée à l'ordinateur-automate que tout s'exprimait en termes de valeurs); ici, on partira d'une considération de bon sens : le seau et la marmite peuvent contenir un nombre entier de patates
- pouvoir simuler l'automate-peleur à l'aide de l'automate-ordinateur (d'abord en pseudo-code, puis en utilisant – enfin – un 'vrai' ordinateur et un 'vrai' langage de programmation)

a) DÉMARCHE : DE L'ALGORITHME ABSTRAIT À L'ALGORITHME CONCRET

- comme pour les dimensions du terrain de RobotProg, on peut imaginer que la contenance du seau et de la marmite (nombre de patates maximum qu'ils peuvent contenir) sont fixées au départ et ne pourront pas varier en cours d'exécution; nous en ferons donc des **constantes**; en spécifiant des valeurs particulières, par exemple

```
Constante contenanceSeau = 15, contenanceMarmite = 33
```

- en cours d'exécution, le seau se vide (*pelerPatate*) et se remplit (*remplirSeau*), tandis que la marmite se remplit (*pelerPatate*) : leur contenu (nombre de patates à un moment donné) est donc **variable**, ce qui nous amène à déclarer :

```
Variable contenuSeau, contenuMarmite : Entier
```

- à quoi correspond la procédure *pelerPatate* ? elle enlève une patate du seau pour l'ajouter (pelée) à la marmite; il faut cette fois deux **affectations** pour arriver à ce résultat :

```

Procédure pelerPatate :
Début
  contenuSeau ← contenuSeau - 1           # une patate en moins
  contenuMarmite ← contenuMarmite + 1     # une patate en plus
Fin

```

- que doit faire la procédure *remplirSeau* ? elle doit mettre dans le seau un nombre de patates égal à la contenance de ce seau, donc, en utilisant l'action d'**affectation**, nous pouvons écrire concrètement :

```

Procédure remplirSeau :
Début
  contenuSeau ← contenanceSeau
Fin

```

⁶⁴ nous utiliserons désormais le terme de « programmable » en lieu et place de celui d'« adaptable »

⁶⁵ le terme d'abstrait fait ici référence au fait que nous ne nous sommes intéressés qu'au fonctionnement de l'algorithme (vu d'un œil extérieur), sans nous préoccuper des détails internes, comme par exemple des données ...

- on se souviendra que le contenu du seau et de la marmite sont inconnus au départ (*c.-à-d. que le nombre de patates qu'ils contiennent peut être compris entre 0 et leur contenance maximale*); pour pouvoir essayer toutes sortes de configurations de départ, nous ajouterons ici une procédure *initialiser* qui permettra d'affecter un nombre de patates au seau et à la marmite ... on peut même imaginer une affectation 'externe', via la procédure *lire()*

```

Procédure initialiser :
Debut
    contenuSeau ← 5
    contenuMarmite ← 15
Fin

```

```

Procédure initialiser :
Debut
    lire(contenuSeau)
    lire(contenuMarmite)
Fin

```

- voilà pour les procédures ... et pour concrétiser les états, comment fait-on ?

c'est assez direct : qu'est-ce qu'un *seauVide* ? (*c'est un seau dont le contenu est 0 patates*) et qu'est-ce qu'une *marmitePleine* ? (*c'est une marmite dont le contenu en nombre de patates est égal à sa contenance*) : la valeur de chacun de ces deux états s'obtient donc par une simple **comparaison** :

seauVide correspond à la comparaison

"est-ce que *contenuSeau* = 0 ?"

marmitePleine correspond à la comparaison

"est-ce que *contenuMarmite* = *contenanceMarmite* ?"

- nous pourrions donc écrire une première version 'concrète' de l'algorithme 'abstrait' de départ :

Algorithme Peleur :

Constante *contenanceSeau* = 15, *contenanceMarmite* = 33

Variable *contenuSeau*, *contenuMarmite* : **Entier**

Déclaration/Définition des procédures

Procédure initialiser :

Debut

contenuSeau ← 5 # combien de patates dans le seau au début

contenuMarmite ← 15 # combien de patates dans la marmite au début

Fin

Procédure remplirSeau :

Debut

contenuSeau ← *contenanceSeau* # mettre toutes les patates possibles

Fin

Procédure pelerPatate :

Debut

contenuSeau ← *contenuSeau* - 1 # une patate de moins

contenuMarmite ← *contenuMarmite* + 1 # une patate de plus

Fin

Algorithme proprement dit : séquence principale

Debut

initialiser

TantQue non *contenuMarmite* = *contenanceMarmite* **Faire**

Si *contenuSeau* = 0 **Alors** *remplirSeau* **FinSi**

pelerPatate

finTQ

Fin.

- un programme d'ordinateur rédigé en langage Pascal est proposé à la page suivante

b) LANGAGE PASCAL (1972)

Comme ce langage fait l'objet d'un cours explicite de programmation, nous renvoyons le lecteur à ce cours et à ses supports ...

```

{ ===== }
{ algo peleur de patates en Pascal (1) }
{ ===== }

program peleurDePatates; { identification du programme/algorithmme }

uses crt; { librairie de fonctions/procédures clavier & écran }

{ === bloc déclaratif de constantes ===== }
const contenanceSeau = 7;
      contenanceMarmite = 43;

{ === bloc déclaratif de variables ===== }
var contenuSeau : integer;
    contenuMarmite : integer;

{ == déclaration et définition des fonctions et procédures ===== }

procedure initialiser; { fixer contenu initial seau et marmite }
begin
    contenuSeau := 5; { affectation }
    contenuMarmite := 15; { affectation }
end;

{ ----- }

procedure remplirSeau; { aller au tas remplir le seau }
begin
    write('seau vide !'); { état 'avant' }
    contenuSeau := contenanceSeau; { affectation 'interne' }
    writeln(' - seau rempli !') { état 'après' }
end;

{ ----- }

procedure pelerPatate; { prendre patate dans seau et mettre dans marmite }
begin
    contenuSeau := contenuSeau - 1; { affectation change état du seau }
    contenuMarmite := contenuMarmite + 1; { idem change état de la marmite }
    writeln('seau : ', contenuSeau, ' - marmite : ', contenuMarmite)
end;

{ === programme principal ===== }

begin
    initialiser; { fixer états initiaux : seau & marmite }
    while not (contenuMarmite = contenanceMarmite) do begin { condition arrêt }
        if (contenuSeau = 0) then remplirSeau; { modifie état du seau }
        pelerPatate { modifie état marmite et seau }
    end;
    writeln('marmite pleine !'); { état final de la marmite }
    readkey
end.

{ ===== }

```

- pas trop mal ... mais nous n'avons pas tout à fait respecté notre contrat de départ : conserver l'algorithme de départ tel quel !

Les états logiques (si jolis parce si 'directs') ont en effet disparu au profit (?) de comparaisons faisant référence explicitement aux données ; ceci signifie une perte d'abstraction ... et d'indépendance (puisque la séquence principale dépend de la manière dont les choses sont implémentées, alors qu'idéalement celles-ci devraient rester 'cachées')

Dans un premier temps, il est possible de récupérer l'abstraction (cacher le détail du *comment ?*) en utilisant des variables logiques et en y affectant – le moment venu (ici dans la procédure *pelerPatate*) – le résultat de la comparaison

(dans la version ci-dessous, nous utiliserons l'affectation 'externe' pour permettre à l'utilisateur de décider du contenu du seau et de la marmite lors de l'initialisation)

Algorithme Peleur :

```

Constante contenanceSeau = 15, contenanceMarmite = 33
Variable contenuSeau, contenuMarmite : Entier
           seauVide, marmitePleine : Logique

Procédure initialiser :
Debut
  lire(contenuSeau)      # combien de patates dans le seau au début
  lire(contenuMarmite)   # combien de patates dans la marmite au début
  marmitePleine ← (contenuMarmite = contenanceMarmite)   # état
  seauVide ← (contenuSeau = 0)                           # état
Fin

Procédure pelerPatate :
Debut
  contenuSeau ← contenuSeau - 1
  contenuMarmite ← contenuMarmite + 1
  marmitePleine ← (contenuMarmite = contenanceMarmite)   # état
  seauVide ← (contenuSeau = 0)                           # état
Fin

...

Debut
  initialiser
  TantQue non marmitePleine Faire
    Si seauVide Alors remplirSeau FinSi
    pelerPatate
  finTQ
Fin.
```

une remarque très importante : en confiant les états à deux variables logiques, il est impératif d'évaluer ces états une première fois dans la procédure d'initialisation : on voit donc les deux mêmes lignes apparaître dans deux procédures différentes (on devrait donc idéalement en faire une procédure à part entière, mais découvrons un outil plus adapté – la fonction – un peu plus loin dans le texte ...)

une version concrète et opérationnelle écrite en Pascal figure à la page suivante

```

{ ===== }
{ algo peleur de patates en Pascal (2) }
{ ===== }

program peleurDePatates; { identification du programme/algorithm }

uses crt; { librairie de fonctions/procédures clavier & écran }

{ === bloc déclaratif de constantes ===== }
const contenanceSeau      = 7;
      contenanceMarmite = 43;

{ === bloc déclaratif de variables ===== }
var contenuSeau      : integer;
    contenuMarmite : integer;
    seauVide        : boolean; { état du seau }
    marmitePleine    : boolean; { état de la marmite }

{ == déclaration et définition des fonctions et procédures == }

procedure initialiser; { fixer contenu initial seau et marmite }
begin
  write('contenu seau au depart ? '); { demander }
  readln(contenuSeau); { affectation 'externe' }
  write('contenu marmite au depart ? '); { demander }
  readln(contenuMarmite); { affectation 'externe' }
  seauVide := (contenuSeau = 0); { état seau }
  marmitePleine := (contenuMarmite = contenanceMarmite); { état marmite }
end;

{ ----- }

procedure remplirSeau; { aller au tas remplir le seau }
begin
  write('seau vide !'); { état 'avant' }
  contenuSeau := contenanceSeau; { affectation 'interne' }
  writeln(' - seau rempli !') { état 'après' }
end;

{ ----- }

procedure pelerPatate; { prendre patate dans seau et mettre dans marmite }
begin
  contenuSeau := contenuSeau - 1; { change état du seau }
  contenuMarmite := contenuMarmite + 1; { change état de la marmite }
  seauVide := (contenuSeau = 0); { état seau }
  marmitePleine := (contenuMarmite = contenanceMarmite); { état marmite }
  writeln('seau : ', contenuSeau, ' - marmite : ', contenuMarmite)
end;

{ ==== programme principal ===== }

begin
  initialiser; { fixer états initiaux : seau & marmite }
  while not marmitePleine do begin { condition arrêt }
    if seauVide then remplirSeau; { modifie état du seau }
    pelerPatate; { modifie état marmite et seau }
  end;
  writeln('marmite pleine !'); { état final de la marmite }
  readkey
end.

{ ===== }

```

Par curiosité, jetons un coup d'œil sur l'implémentation concrète de cette dernière version de l'algorithme avec deux langages 'phares' des années 1955-1965 (émergence des 'vrais' langages de troisième génération : Fortran et Cobol) puis avec un autre langage des années 1970-1980 (à côté de Pascal, déjà illustré aux pages précédentes), le langage C (auquel un autre cours spécifique sera consacré au second semestre)

Pour chacun, nous décrirons rapidement comment ont été implémentés à leur manière les concepts présents dans le langage algorithmique pseudo-code tel que décrit précédemment : commentaires, déclaration des données, constantes et variables (on ne s'intéressera cependant ici qu'aux seuls types entier et logique), instructions d'action (affectations et opérations), expression des conditions, structures alternatives et boucles, procédures

c) LANGAGE FORTRAN (1954)

commentaires

depuis le point d'exclamation (!) à la fin de la ligne

déclaration de données, variables et constantes

- les types entier (INTEGER) et logique (LOGICAL) sont reconnus
- on déclare d'abord le type, puis la liste des variables de ce type
- il n'y a pas de constante à proprement parler, mais il existe la possibilité de variables initialisées auxquelles on peut faire jouer le rôle de pseudo-constantes

```
! *** declaration de variables entières
INTEGER CSEAU, CMARMITE
!
! *** variables entières initialisées (=pseudo-constantes)
INTEGER MSEAU, MMARMITE
DATA MSEAU/17/
DATA MMARMITE/43/
```

affectations et opérations

l'affectation est symbolisée par le signe = (on peut affecter aussi bien des valeurs littérales que le contenu d'autres variables)

```
A = 17 ! affecter la valeur 17 à la variable A
B = A ! affecter le contenu de la variable A à la variable B
```

comme Fortran était destiné à la communauté scientifique, on peut exprimer des expressions numériques aussi facilement qu'en math (symboles +, -, *, /)

```
DELTA = (B * B - 4 * A * C) / 2 * A ! pour équation second degré
```

conditions

les expressions conditionnelles n'utilisent pas (encore) les symboles de comparaison et rappellent les 'mots' de l'assembleur (.EQ., .LT., .LE., ...); par contre, des expressions complexes peuvent être rédigées 'en une fois', à l'aide des opérateurs logiques (.AND., .OR., .NOT.)

```
IF (B.LT.A).AND.(C.NE.B) ! expression : si B<A ET C≠B ...
```

alternatives

à nouveau, on est très proche de l'assembleur, puisque la seule syntaxe possible est : IF condition GOTO ...; il n'y a pas de ELSE explicite (si la condition n'est pas vérifiée, on passe à l'instruction suivante (habituellement elle-même un GOTO))

```
IF (B.LT.A).AND.(C.NE.B) GOTO 225 ! si vrai, aller à ...
GOTO 235 ! sinon, aller à ...
```

il faut donc précéder certaines instructions d'une étiquette (un 'label', un numéro), de manière à pouvoir y faire référence par GOTO

boucles

pas de boucle telle que le Tant que ... ou le Répéter Jusque ... ; il faut, à nouveau comme en assembleur, se débrouiller avec le seul IF ... et le GOTO ...)

procédures

elles existent 'un petit peu' mais leur utilisation nécessite la compréhension de concepts (variables globales et locales) qui ne seront abordés que dans la seconde partie du cours : de toute façon, il faut rédiger un algorithme relativement 'déstructuré' (la caractéristique principale du Fortran n'est certainement pas l'aspect procédural)

procédures d'entrée/sortie (lire et écrire sur les périphériques)

elles aussi sont rudimentaires (elles font un peu penser à ce qu'on retrouvera 25 ans plus tard en langage C !); les instructions lire() et écrire() s'écrivent logiquement et respectivement READ() et WRITE(), avec entre parenthèses

- un premier nombre qui fait référence au périphérique (5 = code du périphérique d'entrée standard : lecteur de cartes perforées ou clavier, 6 = imprimante ou écran)
- un second nombre qui fait référence à une ligne numérotée du programme et qui contient le FORMAT des données lues ou écrites

```
! *****
! *** programme peleur de patates en Fortran ***
! *****
!
!      PROGRAM PELEUR
!
! *** variables initialisées (=pseudo-constantes)
!
!      INTEGER MSEAU, MMARMITE           ! contenance seau & marmite
!      DATA MSEAU/17/, MMARMITE/43/    ! valeurs
!
! *** déclaration des variables
!
!      INTEGER CSEAU, CMARMITE           ! contenu seau & marmite
!      LOGICAL SEAUVI, MARMPL           ! états : seauVide & marmitePleine
!
! *** formatage des entrées/sorties
!
11      FORMAT('contenu du seau au depart ? ')
12      FORMAT('contenu de la marmite au depart ? ')
13      FORMAT('seau=',I2,' - marmite=',I2)
14      FORMAT('seau vide >>> seau rempli !')
15      FORMAT('marmite pleine, fini !')
21      FORMAT(I2)                       ! entier sur 2 chiffres
!
! *** programme principal *****
!
!      WRITE(6,11)                       ! message d'invite
!      READ(5,21) CSEAU                  ! affectation 'externe'
!      WRITE(6,12)                       ! message d'invite
!      READ(5,21) CMARMITE               ! affectation 'externe'
!
! *** programmation 'spaghetti' avec if ... et goto ...
!
100     SEAUVI = CSEAU.EQ.0               ! état : seau vide ?
!      MARMPL = CMARMITE.EQ.MMARMITE     ! état : marmite pleine ?
!      IF (MARMPL) GOTO 900              ! si fini, stop et fin
!      IF (SEAUVI) GOTO 300              ! sinon: si seau vide, remplir
200     CSEAU = CSEAU - 1                 ! sinon: peler (1)
!      CMARMITE = CMARMITE + 1            ! peler (2)
!      WRITE(6,13) CSEAU, CMARMITE        ! message
!      GOTO 100                          ! boucler
300     CSEAU = MSEAU                    ! remplir seau
!      WRITE(6,14)                       ! message
!      GOTO 200                          ! retourner peler
!
900     WRITE(6,15)                      ! message de fin
!      END
!
! *****
```

comme on peut le voir ci-dessous, forcer une écriture procédurale en Fortran possède un réel coût rédactionnel (plutôt qu'à des procédures, il faudrait faire appel à des fonctions ... mais ceci sort du cadre de cette introduction ...)

```

! *****
! *** programme peleur en Fortran(2) ***
! *****
!
      PROGRAM PELEUR
!
! *** declaration des variables
      INTEGER CSEAU, CMARMITE, MSEAU, MMARMITE
      DATA MSEAU/17/, MMARMITE/43/
!
! *** globalisation des variables
      COMMON CSEAU, CMARMITE, MSEAU, MMARMITE
!
! *** programme principal *****
!
      CALL INIT
100    IF (CMARMITE.EQ.MMARMITE) GOTO 900      ! comparaison + stop
      IF (CSEAU.EQ.0) CALL REMPLIR
      CALL PELER
      GOTO 100                                ! boucle 'tant que'
!
900    WRITE(6,15)                            ! message et fin
15     FORMAT('marmite pleine, fini !')
      END
!
! *****
!
! *** procedures
!
      SUBROUTINE INIT
      INTEGER CSEAU, CMARMITE, MSEAU, MMARMITE      ! déclaration
      COMMON CSEAU, CMARMITE, MSEAU, MMARMITE      ! partage
11     FORMAT('contenu du seau au depart ? ')
12     FORMAT('contenu de la marmite au depart ? ')
21     FORMAT(I2)                                ! format de donnée : entier 2 chiffres
      WRITE(6,11)                                ! invite
      READ(5,21) CSEAU                            ! affectation 'externe'
      WRITE(6,12)                                ! invite
      READ(5,21) CMARMITE                         ! affectation 'externe'
      END
!
      SUBROUTINE REMPLIR
      INTEGER CSEAU, CMARMITE, MSEAU, MMARMITE      ! déclaration
      COMMON CSEAU, CMARMITE, MSEAU, MMARMITE      ! partage
      CSEAU = MSEAU
      WRITE(6,13)                                ! message
13     FORMAT('seau vide >>> seau rempli !')
      END
!
      SUBROUTINE PELER
      INTEGER CSEAU, CMARMITE, MSEAU, MMARMITE      ! déclaration
      COMMON CSEAU, CMARMITE, MSEAU, MMARMITE      ! partage
      CSEAU = CSEAU - 1
      CMARMITE = CMARMITE + 1
      WRITE(6,14) CSEAU,CMARMITE                  ! affichage 2 variables
14     FORMAT('seau=',I2,' - marmite=',I2)
      END
!
! *****

```


d) LANGAGE COBOL (1959)

commentaires

seule possibilité, une ligne entière commençant par une étoile (*) comme septième caractère (pas moyen de placer des commentaires en fin de ligne, donc ... on estime que la rédaction d'un programme dans ce langage est tellement 'littéraire' et proche de l'anglais que les commentaires sont relativement peu nécessaires)

déclaration de données, variables et constantes

- le type entier est reconnu, aussi bien sous forme décimale que sous forme interne (binaire)
- les variables sont déclarées à l'aide d'une syntaxe qui comporte un aspect structurel (77 = variable scalaire), un nom (avec tiret éventuel), un type (annoncé par PIC = picture) et une longueur (99 = entier sur 2 chiffres)

```
77 CONTENU-SEAU PIC 99.
```

- il n'y a pas de constante à proprement parler, mais il existe la possibilité de variables initialisées auxquelles on peut faire jouer le rôle de pseudo-constantes (clause VALUE)

```
77 CONTENANCE-SEAU PIC 99 VALUE 17.
```

- il n'y a pas de type logique à proprement parler, mais à toute variable scalaire (77), on peut associer une ou plusieurs pseudo-variables logiques (88 et VALUE) utilisables directement dans les conditions et dont l'affectation de la valeur vrai/faux est entièrement automatique

```
77 CONTENU-SEAU PIC 99.
88 SEAU-VIDE VALUE 0.
77 CONTENU-MARMITE PIC 99.
88 MARMITE-PLEINE VALUE 43.
```

(SEAU-VIDE est vrai si CONTENU-SEAU = 0, est faux sinon ... on pourra ainsi écrire IF SEAU-VIDE ...)

affectations et opérations

Cobol est réputé verbeux (il est en cela diamétralement opposé au Fortran, qui utilise les symboles mathématiques) parce qu'il utilise des verbes (anglais) pour toutes les opérations (rappelant au passage les 'mots' de l'assembleur)

```
MOVE 17 TO A      # A ← 17
MOVE A TO B       # B ← A
ADD A TO B GIVING C # C ← A + B
```

fort heureusement, pour des expressions plus compliquées, il y a le verbe COMPUTE

```
COMPUTE DELTA = (B * B - 4 * A * C) / 2 * A
```

conditions

les expressions conditionnelles utilisent les symboles de comparaison; des expressions complexes peuvent être rédigées 'en une fois', à l'aide des opérateurs logiques (AND, OR, NOT)

```
IF B <= A AND C NOT= B      # expression : si B<A ET C≠B ...
```

procédures

Cobol est fondamentalement procédural : un programme est un ensemble de 'paragraphes' qui sont en fait des séquences nommées et que l'on invoque (exécute) à l'aide du verbe PERFORM

alternatives

la syntaxe de base est : *IF condition THEN PERFORM paragraphe ...*; il n'y a pas de ELSE (du moins pas avant la révision du langage en 1985)

```
IF C = 0 THEN PERFORM TRT-ERREUR # si vrai, exécuter ...
```

boucles

Cobol propose une 'vraie' instruction de boucle correspondant au Tant que ... (avec test antérieur, donc principe de précaution); malheureusement elle s'écrit *PERFORM paragraph UNTIL condition* (mais signifie en fait *TantQue non condition Faire paragraph*, ce qui a toujours fait hurler de rire les détracteurs de ce langage ☹)

procédures d'entrée/sortie (lire et écrire sur les périphériques)

Cobol n'est pas un langage destiné à être 'conversationnel'; il existe des instructions lire() et écrire() qui s'écrivent respectivement ACCEPT et DISPLAY, et permettent au programme d'interagir avec l'opérateur à la console (p.ex. pour demander le montage d'une bande magnétique)

en examinant l'exemple ci-dessous, on constate qu'il correspond point pour point à notre algorithme idéal écrit en pseudo-code (pour rappel, on est à l'aube des années 1960), et cela grâce aux pseudo variables logiques (les 88) et aux paragraphes (procédures); le programme présente les deux qualités fondamentales qu'on attend : bon niveau d'abstraction et indépendance à la manière dont les données sont implémentées

```

*****
*** programme peleur de patates en Cobol ***
*****

IDENTIFICATION DIVISION.
PROGRAM-ID.                                PELEUR.

*** déclaration des données *****

DATA DIVISION.
WORKING-STORAGE SECTION.
77 CONTENANCE-SEAU                        PIC 99 VALUE 17.
77 CONTENU-SEAU                          PIC 99.
88 SEAU-VIDE                             VALUE 0.
77 CONTENU-MARMITE                        PIC 99.
88 MARMITE-PLEINE                         VALUE 43.

*** programme principal *****

PROCEDURE DIVISION.
MAIN SECTION.
    PERFORM INITIALISER.
    PERFORM BOUCLE UNTIL MARMITE-PLEINE.
    DISPLAY "marmite pleine : fini !".
    EXIT PROGRAM.
    STOP RUN.

*** procédures *****

INITIALISER.
    DISPLAY "contenu seau au depart ? ".
    ACCEPT CONTENU-SEAU.
    DISPLAY " contenu marmite au depart ?".
    ACCEPT CONTENU-MARMITE.

BOUCLE.
    IF SEAU-VIDE PERFORM REMPLIR-SEAU.
    PERFORM PELER-PATATE.

REMPLIR-SEAU.
    MOVE CONTENANCE-SEAU TO CONTENU-SEAU.
    DISPLAY "seau vide >>> seau rempli !".

PELER-PATATE.
    ADD 1 TO CONTENU-MARMITE.
    SUBTRACT 1 FROM CONTENU-SEAU.
    DISPLAY "seau=" CONTENU-SEAU " - marmite=" CONTENU-MARMITE.

*****

```

e) LANGAGE C (1973)

commentaires

depuis // jusqu'à la fin de la ligne ou entre /* et */

déclaration de données, variables et constantes

- le type entier est reconnu, mais pas le type logique (tout se fait à travers le type entier avec la convention que 0 correspond à faux et toute autre valeur à vrai)
- les variables sont déclarées un peu comme en Fortran : le type d'abord et la liste des variables de ce type ensuite

```
int contenuSeau, contenuMarmite;
```

- il n'y a pas de constante à proprement parler, mais une directive qui rappelle les `def` de l'assembleur : on associe un nom symbolique et une valeur; durant la compilation, toutes les occurrences du nom sont remplacées par la valeur

```
#define CONTENANCE_SEAU 17
#define CONTENANCE_MARMITE 43
```

affectations et opérations

l'affectation utilise le symbole `=` et (comme en Fortran) des expressions (numériques) peuvent être écrites en une seule ligne (avec les opérateurs `+`, `-`, `*`, `/`)

des 'sucres syntaxiques'⁶⁶ permettent de compacter des instructions utilisées fréquemment, p.ex.

```
--contenuSeau;           // contenuSeau ← contenuSeau - 1
++contenuMarmite;        // contenuMarmite ← contenuMarmite + 1
```

conditions

les expressions conditionnelles utilisent les symboles de comparaison (attention l'égalité utilise `==`); des expressions complexes peuvent être rédigées 'en une fois', à l'aide des opérateurs logiques (AND, OR, NOT)

procédures

C encourage l'écriture procédurale (mais les procédures sont des 'fonctions' spéciales) : en C tout est fonction, y compris la séquence principale baptisée obligatoirement `main`)

alternatives

C propose l'alternative complète IF (condition) { ... } ELSE { ... }

```
if (contenuSeau == 0) remplirSeau();
```

boucles

fondamentalement, C ne propose qu'une seule boucle (Tant que ..., bien entendu, principe de précaution oblige) mais la décline sous différentes formes (y compris le Répéter ... Jusque ...)

procédures d'entrée/sortie (lire et écrire sur les périphériques)

C comporte une vaste librairie de fonctions d'entrée/sortie; dans l'exemple de la page suivante, les procédures `lire()` et `écrire()` sont implémentées en invoquant respectivement `scanf()` et `printf()`; elles rappellent un peu les fonctions du Fortran, à ceci près qu'elles 'embarquent' leur formatage

⁶⁶ http://fr.wikipedia.org/wiki/Sucre_syntaxique

```

// *****
// *** programme peleur de patates en C ***
// *****

#include <stdio.h>    // librairies de fonctions d'entrée-sortie

// constantes symboliques

#define CONTENANCE_SEAU 17
#define CONTENANCE_MARMITE 43

// prototype des procédures

void initialiser();
void remplirSeau();
void pelerPatate();

// variables (globales)

int contenuSeau, contenuMarmite;

// *** fonction principale *****

int main() {
    initialiser();
    while (! (contenuMarmite == CONTENANCE_MARMITE)) {    // tant que
        if (contenuSeau == 0) remplirSeau();
        pelerPatate();
    }
    printf("\nmarmite pleine : fini !");
    return 0;
}

// *****

// définition des procédures

void initialiser() {
    printf("contenu marmite au depart ? ");    // invite
    scanf("%d", &contenuMarmite);    // affectation 'externe'
    printf("contenu seau au depart ? ");    // invite
    scanf("%d", &contenuSeau);    // affectation 'externe'
}

// -----

void pelerPatate() {
    --contenuSeau;    // décrémenter seau
    ++contenuMarmite;    // incrémenter marmite
    printf("seau=%d : marmite=%d \n", contenuSeau, contenuMarmite);
}

// -----

void remplirSeau() {
    contenuSeau = CONTENANCE_SEAU;    // affectation interne
    printf("seau vide ! >>> seau rempli ! \n\n");
}

// *****

```

f) EXERCICES

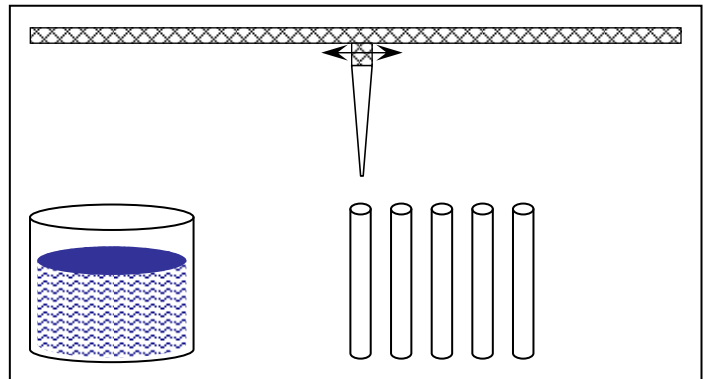
Interrogation de novembre (année académique 2011-2012)

Une unité de production pharmaceutique dispose sous atmosphère stérile d'un robot dont le travail est de remplir automatiquement des ampoules de médicament.

Pour ce faire, il dispose d'une cuve de stockage, d'une pipette accrochée à un rail et pouvant se déplacer latéralement, et d'un lot d'ampoules identiques en nombre indéterminé.

Au démarrage, la cuve contient du produit, toutes les ampoules sont vides et ouvertes, la pipette est vide et est positionnée au-dessus de la première ampoule (à gauche de la série, cfr. ci-contre).

Le but est de remplir toutes les ampoules mais à concurrence du contenu de la cuve



Le robot est livré avec les actions et les états suivants (la cuve, les ampoules et la pipette contiennent un nombre entier de gouttes, mais on ne sait rien quant aux contenances relatives cuve/pipette/ampoule) :

Actions 'élémentaires'

remplirPipette	Le robot quitte sa position courante, va à la cuve, remplit la pipette et se repositionne au-dessus de la <u>première</u> ampoule (à gauche)
atteindreAmpouleSuivante	Le robot déplace la pipette et la positionne au-dessus de l'ampoule suivante (de gauche à droite)
verserGoutte	Le robot presse la pipette : une goutte de produit passe de la pipette à l'ampoule

États logiques

ampouleRemplie	vrai si l'ampoule correspondant à la position courante est remplie, faux sinon
pipetteVide	vrai si la pipette est vide, faux sinon
lotTerminé	vrai si <u>toutes</u> les ampoules sont remplies, faux sinon
cuveVide	vrai si la cuve est vide, faux sinon

Compte tenu des conditions initiales favorables (tous les états de départ étant connus), l'analyste en charge du dossier a rédigé l'algorithme "abstrait optimiste" suivant :

Algorithme remplisseur :

Début

```

remplirPipette           # puisqu'elle est vide et pas la cuve
Répéter                 # boucle principale 'optimiste'
  Répéter               # vider la pipette remplie
    verserGoutte         # puisque la pipette n'est pas vide
    Si ampouleRemplie Alors # c'est possible après avoir versé
      atteindreAmpouleSuivante # et c'est la seule chose à faire
    finSi
  Jusque pipetteVide ou lotTerminé # les deux seules possibilités
  Si pipetteVide et non (cuveVide ou lotTerminé) Alors # ici trois conditions !
    remplirPipette
    tantQue ampouleRemplie Faire # se repositionner ...
      atteindreAmpouleSuivante # ... sur la première ampoule non remplie
    finTQ
  finSi
Jusque lotTerminé ou cuveVide # deux possibilités d'arrêt

```

Fin

- Rédigez en pseudo-code un algorithme procédural "concret" équivalent en déclarant et en utilisant un ensemble approprié de données typées (*indication : utilisez une variable de position*)
- L'analyste a oublié que si le robot s'arrête lorsque la cuve est vide, il se pourrait qu'il reste alors dans la pipette suffisamment de produit pour remplir toutes les ampoules et mener à bien la mission principale. Essayez d'inclure cette situation dans votre version

9.2. NOTION DE FONCTION

Chapitre très introductif, histoire d'être complet à la fin de cette première partie ...

a) PRINCIPE

Afin d'éviter une référence explicite aux données dans l'algorithme principal :

```
TantQue non contenuMarmite = contenanceMarmite Faire
  Si contenuSeau = 0 Alors remplirSeau FinSi
  pelerPatate
finTQ
```

nous sommes passés par une phase déclarative des états via des variables logiques (booléennes) :

Variable **seauVide, marmitePleine** : **Logique**

et de leur affectation dans la procédure qui modifie les états correspondants :

```
Procédure pelerPatate :
Debut
  contenuSeau ← contenuSeau - 1
  contenuMarmite ← contenuMarmite + 1
  marmitePleine ← (contenuMarmite = contenanceMarmite)
  seauVide ← (contenuSeau = 0)
Fin
```

Mais on peut estimer que l'on fait faire ici à cette procédure quelque chose qui n'est pas de sa responsabilité : elle ne doit 'que' peler une patate et non s'interroger (ou fournir une information) sur l'état de la marmite et/ou du seau.

C'est ici qu'intervient un nouvel outil algorithmique : la fonction.

Sans entrer ici dans le détail (cfr. seconde partie du cours), on considèrera que la procédure et la fonction sont deux outils de généralisation :

la procédure est une généralisation du concept d'action

c'est une séquence nommée d'instructions, permettant d'abstraire un traitement en en cachant les détails ... voir l'exemple du Peleur

la fonction est une généralisation du concept de valeur

c'est également une séquence nommée d'instructions, mais son but est de calculer une valeur et de la mettre à la disposition de l'algorithme

Il est courant d'utiliser pour introduire le concept de fonction (d'un abord plus difficile que la procédure) l'image mathématique de la fonction : quand on utilise une calculatrice pour évaluer $\sin(17^\circ)$, ou $e^{3.5}$, on s'attend à ce qu'une (seule !) valeur soit calculée et 'affichée' (ce qui se passe à l'intérieur de la calculatrice peut rester mystérieux)

La fonction en algorithmique (et en programmation) fait de même : c'est une 'boîte noire' qui calcule et renvoie une et une seule valeur ; la manière dont on la déclare est assez semblable à la procédure à quelques détails essentiels près :

1) sa déclaration

- commence par le mot **Fonction**
- spécifie son **nom** suivi de deux parenthèses
- déclare le **type de la valeur** calculée

```
Fonction seauVide() : Logique
Début
  renvoie (contenuSeau = 0)
Fin
```

2) sa séquence

- comporte les marqueurs habituels de début et de fin
- utilise l'instruction **renvoie** à laquelle on associe le **calcul** de la valeur à renvoyer

Dans l'exemple ci-dessus, il faut donc l'interpréter comme suit : la fonction `seauVide()` compare le contenu du seau à la valeur 0 : cette comparaison est une valeur logique (vrai/faux) renvoyée par la fonction

- nous pouvons donc écrire une dernière version 'concrète' pseudo-code de l'algorithme 'abstrait' de départ :

Algorithme Peleur :

Constante `contenanceSeau = 15, contenanceMarmite = 33`

Variable `contenuSeau, contenuMarmite : Entier`

déclarations/définitions de fonctions

Fonction `seauVide()` : Logique

Debut

 renvoie (`contenuSeau = 0`)

Fin

Fonction `marmitePleine()` : Logique

Debut

 renvoie (`contenuMarmite = contenanceMarmite`)

Fin

déclarations/définitions de procédures

Procédure `initialiser` :

Debut

 lire(`contenuSeau`) # combien de patates dans le seau au début

 lire(`contenuMarmite`) # combien de patates dans la marmite au début

Fin

Procédure `remplirSeau` :

Debut

`contenuSeau ← contenanceSeau` # mettre toutes les patates possibles

Fin

Procédure `pelerPatate` :

Debut

`contenuSeau ← contenuSeau - 1`

`contenuMarmite ← contenuMarmite + 1`

Fin

algorithme proprement dit : séquence principale

Debut

`initialiser`

 TantQue non `marmitePleine()` Faire

 Si `seauVide()` Alors `remplirSeau` FinSi

`pelerPatate`

 finTQ

Fin.

On remarquera que les variables logiques qui permettaient d'évaluer et de mémoriser les états ont disparu et ont été remplacées par des fonctions équivalentes : désormais, chacun des outils (procédure, fonction) 'enferme et cache' les détails de ce qu'il a à faire ... et il fait ce qu'il a à faire, et rien d'autre !

Les pages suivantes contiennent quelques exemples d'implémentation dans des langages qui proposent des fonctions en particulier Pascal et C

ensuite, nous concluons cette première partie du cours avec une rédaction de l'algorithme 'Peleur' à l'aide de langages plus récents; attention, certains de ces langages (Python et Ruby notamment) sont des langages de programmation 'orientés objet'; toutefois nous n'utiliserons ici que les seules possibilités classiques de la programmation impérative, procédurale et itérative de ces langages, le but restant d'illustrer le passage de l'algorithme au programme ... et de faire voir différents aspects – notamment syntaxiques – de différents langages de programmation à partir de cas simples mais concrets.

b) LANGAGE PASCAL

```

{ ===== }
{ algo peleur de patates en Pascal (3) }
{ ===== }

program peleurDePatates; { identification du programme/algorithmes }

uses crt; { librairie de fonctions/procédures clavier & écran }

{ === bloc déclaratif de constantes ===== }
const contenanceSeau = 7;
      contenanceMarmite = 43;

{ === bloc déclaratif de variables ===== }
var contenuSeau : 0..contenanceSeau;
    contenuMarmite : 0..contenanceMarmite;

{ == déclaration et définition des fonctions et procédures == }

function seauVide : boolean; { seau vide ? oui/non }
begin
    seauVide := (contenuSeau = 0) { comparer }
end;

{ ----- }

function marmitePleine : boolean; { marmite pleine ? oui/non }
begin
    marmitePleine := (contenuMarmite = contenanceMarmite) { comparer }
end;

{ ----- }

procedure initialiser; { fixer contenu initial seau et marmite }
begin
    write('contenu seau au depart ? '); { demander }
    readln(contenuSeau); { affectation 'externe' }
    write('contenu marmite au depart ? '); { demander }
    readln(contenuMarmite); { affectation 'externe' }
end;

{ ----- }

procedure remplirSeau; { aller au tas remplir le seau }
begin
    write('seau vide !'); { état 'avant' }
    contenuSeau := contenanceSeau; { affectation 'interne' }
    writeln(' - seau rempli !') { état 'après' }
end;

{ ----- }

procedure pelerPatate; { prendre patate dans seau et mettre dans marmite }
begin
    contenuSeau := contenuSeau - 1; { change état du seau }
    contenuMarmite := contenuMarmite + 1; { change état de la marmite }
    writeln('seau : ', contenuSeau, ' - marmite : ', contenuMarmite)
end;

{ ===== programme principal ===== }

begin
    initialiser; { fixer états initiaux : seau & marmite }
    while not marmitePleine do begin { condition arrêt = marmite est pleine }
        if seauVide then remplirSeau; { modifie état du seau }
        pelerPatate { modifie état de la marmite }
    end;
    writeln('marmite pleine !'); { état final de la marmite }
    readkey
end.

{ ===== }

```


c) LANGUAGE C

```

// *****
// *** programme peleur de patates en C (2) ***
// *****

#include <stdio.h>

// constantes symboliques
#define CONTENANCE_SEAU 17
#define CONTENANCE_MARMITE 43

// prototype des procédures
void initialiser();
void remplirSeau();
void pelerPatate();

// prototype des fonctions
int seauVide();
int marmitePleine();

// variables (globales)
int contenuSeau;
int contenuMarmite;

// *** fonction principale *****

int main() {
    initialiser();
    while (! marmitePleine()) {
        if (seauVide()) remplirSeau();
        pelerPatate();
    }
    printf("\nmarmite pleine : fini !");
    return 0;
}
// *****

// définition des procédures

void initialiser() {
    printf("contenu marmite au depart ? ");
    scanf("%d", &contenuMarmite);
    printf("contenu seau au depart ? ");
    scanf("%d", &contenuSeau);
}

void pelerPatate() {
    contenuSeau--;
    contenuMarmite++;
    printf("seau=%d : marmite=%d \n", contenuSeau, contenuMarmite);
}

void remplirSeau() {
    contenuSeau = CONTENANCE_SEAU;
    printf("seau vide ! >>> seau rempli ! \n\n");
}

// définition des fonctions

int seauVide() {
    return (contenuSeau == 0);
}

int marmitePleine() {
    return (contenuMarmite == CONTENANCE_MARMITE);
}

// *****

```

d) LANGAGE PERL (1987)

```
#!/usr/bin/perl
use strict;

# déclaration de constantes #

use constant contenanceMarmite => 43;
use constant contenanceSeau    => 17;

# déclaration de variables (globales à l'aide du symbole $) #

my $contenuMarmite;
my $contenuSeau;

# *** fonctions ***** #

sub seauVide {
    $contenuSeau == 0;          # renvoie la valeur (V/F) de la comparaison #
}

sub marmitePleine {
    $contenuMarmite == contenanceMarmite; # idem : comparaison #
}

# *** procédures ***** #

sub initialiser {
    print "contenu marmite au depart ? "; # message d'invite #
    $contenuMarmite = <STDIN>;             # affectation 'externe' #
    print "contenu seau au depart ? ";    # message d'invite #
    $contenuSeau = <STDIN>;                # affectation 'externe' #
}

sub remplirSeau {
    $contenuSeau = contenanceSeau;        # affectation d'une constante #
    print "seau vide >> seau rempli !\n";
}

sub pelerPatate {
    $contenuSeau--;                      # décrémente (-1) contenu seau
    $contenuMarmite++;                   # incrémente (+1) contenu marmite
    print "seau=$contenuSeau - marmite=$contenuMarmite\n";
}

# *** programme principal ***** #

initialiser;
while (!marmitePleine) {
    if (seauVide) { remplirSeau; }
    pelerPatate;
}
print "marmite pleine, fini !\n"

# ***** #
```

e) LANGAGE PYTHON (1991)

```
#!/usr/bin/env python

# déclaration de constantes #

CONTENANCE_MARMITE = 43
CONTENANCE_SEAU = 17

# *** fonctions ***** #

def seauVide():
    global contenuSeau
    return contenuSeau == 0

def marmitePleine():
    global contenuMarmite
    return contenuMarmite == CONTENANCE_MARMITE

# *** procédures ***** #

def initialiser():
    global contenuSeau
    global contenuMarmite
    contenuSeau = input("contenu seau au depart ? ")
    contenuMarmite = input("contenu marmite au depart ? ")

def pelerPatate():
    global contenuSeau
    global contenuMarmite
    contenuSeau = contenuSeau - 1
    contenuMarmite = contenuMarmite + 1
    print "seau=", contenuSeau, ": marmite=", contenuMarmite

def remplirSeau():
    global contenuSeau
    contenuSeau = CONTENANCE_SEAU
    print "seau vide ! >>> seau rempli !"

# *** programme principal ***** #

initialiser()
while not marmitePleine():
    if seauVide(): remplirSeau()
    pelerPatate()
print "marmite pleine : fini !"

# ***** #
```

f) LANGAGE RUBY (1995)

```
#!/usr/bin/env ruby

# déclaration de constantes #

CONTENANCE_MARMITE = 43
CONTENANCE_SEAU = 17

# *** fonctions ***** #

def seauVide?
  $contenuSeau == 0
end

def marmitePleine?
  $contenuMarmite == CONTENANCE_MARMITE
end

# *** procédures ***** #

def initialiser
  puts "contenu marmite au depart ? "
  $contenuMarmite = gets.to_i
  puts "contenu seau au depart ? "
  $contenuSeau = gets.to_i
end

def remplirSeau
  $contenuSeau = CONTENANCE_SEAU
  puts "seau vide ! >>> seau rempli !"
end

def pelerPatate
  $contenuSeau -= 1
  $contenuMarmite += 1
  puts "seau=#{$contenuSeau} : marmite=#{$contenuMarmite}"
end

# *** programme principal ***** #

initialiser
until marmitePleine?
  remplirSeau if seauVide?
  pelerPatate
end
puts "\nmarmite pleine : fini !\n"

# ***** #
```

9.3. EXERCICES

- 1) Donner une version concrète (avec données) de l'automate 'Peleur' proposant un état logique supplémentaire : <tasVide> (exprimant le caractère non inépuisable du tas de patates)

Procédures	description
remplirSeau	<i>prend le seau, va au tas et ramène un seau de patates</i>
pelerPatate	<i>prend une patate dans le seau, pèle la patate, dépose la patate pelée dans la marmite</i>

États	description
marmitePleine	<i>vrai quand la marmite est pleine, faux sinon</i>
seauVide	<i>vrai si le seau ne contient aucune patate, faux sinon</i>
tasVide	<i>vrai s'il n'y a plus de patate sur le tas, faux sinon</i>

- on conserve toutes les autres 'inconnues' de départ (taille relative seau/marmite, état du seau, état de la marmite, état du tas au départ, taille relative tas/marmite)
- 2) Donner des versions concrètes adaptées de cet automate (même mission), mais avec des conditions initiales différentes :
- le seau et la marmite sont vides au départ, état du tas inconnu
 - le seau et la marmite sont vides au départ, tas suffisant pour la marmite
 - le seau n'est pas vide, la marmite est vide, le tas insuffisant

- 3) Donner une version concrète (avec données) de l'automate 'Trieur' dont la mission (§ 6.4) est de « *mettre toutes les pièces du tas dans les bacs respectifs* »

Procédures	description
prendrePièce	<i>aller au tas et prendre une pièce avec la pince</i>
déposerBac1	<i>va au bac de 0.1€ et y lâche la pièce qui est dans la pince</i>
déposerBac2	<i>va au bac de 0.2€ et y lâche la pièce qui est dans la pince</i>
déposerBac5	<i>va au bac de 0.5€ et y lâche la pièce qui est dans la pince</i>

États	description
tasVide	<i>vrai quand le tas ne contient plus de pièce, faux sinon</i>
pièceDe1	<i>vrai si la pièce dans la pince vaut 0.1€, faux sinon</i>
pièceDe2	<i>vrai si la pièce dans la pince vaut 0.2€, faux sinon</i>
pièceDe5	<i>vrai si la pièce dans la pince vaut 0.5€, faux sinon</i>

- rien n'est spécifié quant à l'état du tas au départ (vide ou pas vide) ni du nombre et de la proportion des pièces de 0.1, 0.2 et 0.5€ dans ce tas
 - de même, l'état des bacs récepteurs est inconnu (il suffit d'ailleurs de jeter un œil sur les états proposés pour s'apercevoir que cela n'intervient pas dans le problème)
- 4) Donner des versions concrètes adaptées de cet automate (même mission), mais avec des conditions initiales différentes :
- la pince est vide au départ (état du tas inconnu)
 - la pince contient une pièce au départ (état du tas inconnu)
 - le tas n'est pas vide au départ (état de la pince inconnu)
 - le tas est vide au départ (état de la pince inconnu)
 - le tas n'est pas vide et la pince est vide au départ

- 5) Faire de même avec l'automate 'Remplisseur' (§ 6.4) : imaginez l'un ou l'autre scénario plausible

- 6) Soit l'algorithme concret du Peleur de Patates suivant :

Algorithme Peleur :

Constante contenanceSeau = 15, contenanceMarmite = 33

Variable contenuSeau, contenuMarmite : **Entier**

Algorithme : séquence principale

Debut

```

contenuSeau ← 5           # combien de patates dans le seau au début
contenuMarmite ← 15       # combien de patates dans la marmite au début
TantQue non contenuMarmite = contenanceMarmite Faire
  Si contenuSeau = 0 Alors
    contenuSeau ← contenanceSeau      # mettre les patates possibles
  FinSi
  contenuSeau ← contenuSeau - 1      # une patate de moins
  contenuMarmite ← contenuMarmite + 1 # une patate de plus
finTQ

```

Fin.

soit également le langage-machine (instructions à une opérande) évoqué dans ce chapitre :

code	nature opérande	exemple	description
10	valeur	1009	mettre la valeur 9 dans l'accumulateur
11	adresse-mémoire	1152	mettre le contenu de la cellule-mémoire n°52 dans l'accumulateur
20	valeur	2011	ajouter la valeur 11 au contenu de l'accumulateur
21	adresse-mémoire	2153	ajouter le contenu de la cellule-mémoire n°53 à l'accumulateur
30	valeur	3003	soustraire la valeur 3 du contenu de l'accumulateur
31	adresse-mémoire	3155	soustraire le contenu de la cellule-mémoire n°55 de l'accumulateur
41	adresse-mémoire	4160	copier le contenu de l'accumulateur dans la cellule-mémoire n°60
51	adresse-mémoire	5112	mettre l'adresse 12 dans le compteur ordinal
60	valeur	6038	comparer l'égalité du contenu de l'accumulateur et de la valeur 38
61	adresse-mémoire	6017	comparer l'égalité du contenu de l'accumulateur et du contenu de la case-mémoire d'adresse 17
00	valeur 00	0000	arrêter le programme

rédigez cet algorithme au moyen de ce langage-machine

(les cases-mémoire d'adresses 00 à 09 sont disponibles pour y stocker les données (constantes et variables), le code du programme occupera les adresses consécutives à partir de 10)

- 7) prendre ensuite une version plus procédurale de cet algorithme concret (p.96) et en donner une version procéduralement équivalente en langage-machine

10. EXERCICES DE SYNTHÈSE

▪ Une mission RobotProg : trajet minimum (1)

Objectif à atteindre :

Amener le robot face au mur le plus proche par rapport à sa position initiale

Conditions initiales :

Terrain :	quelconque, sans obstacle
Position du robot :	quelconque
Direction du robot :	quelconque

Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite
États logiques :	murEnFace
États numériques :	distanceMur, dxRobot, dyRobot
Instructions :	tantQue, Si Alors Sinon, affectation

Consignes particulières :

- Résoudre cette mission en utilisant votre procédure allerAuMur et en écrivant de nouvelles procédures spécialisées et réutilisables

Indications utiles :

- --- aucune ---

Variantes intéressantes :

- --- aucune ---

▪ Une mission RobotProg : trajet minimum (2)

Objectif à atteindre :

Amener le robot face au coin le plus proche par rapport à sa position initiale

Conditions initiales :

Terrain :	quelconque, sans obstacle
Position du robot :	quelconque
Direction du robot :	quelconque

Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite
États logiques :	murEnFace
États numériques :	distanceMur, dxRobot, dyRobot
Instructions :	tantQue, Si Alors Sinon, affectation

Consignes particulières :

- comme précédemment

Indications utiles :

- --- aucune ---

Variantes intéressantes :

- --- aucune ---

▪ Une mission RobotProg : le robot-peintre (1)

Objectif à atteindre :

Peindre l'entièreté du terrain, par lignes successives

Conditions initiales :

Terrain :	quelconque, sans obstacle
Position du robot :	dans un coin (quelconque)
Direction du robot :	quelconque, mais pas face à un mur

Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite, marquer
États logiques :	murEnFace, murAGauche, murADroite
États numériques :	--- aucun ---
Instructions :	tantQue, Si Alors Sinon, affectation

Consignes particulières :

- aucune, sinon la modularité ! ---

Indications utiles :

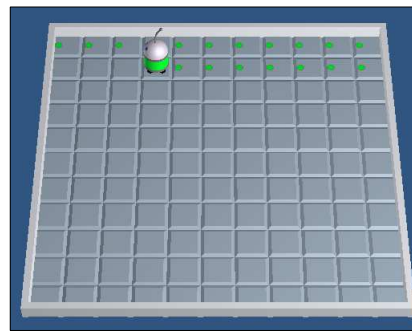
- aucune ---

Variations possibles :

- Que faut-il modifier à votre solution pour la modification suivante des conditions initiales ?

Direction du robot :	quelconque (même face à un mur)
----------------------	---------------------------------

- Votre solution est-elle générale ? l'avez-vous testée avec un terrain possédant un nombre pair de lignes, un nombre impair de lignes, idem pour les colonnes ?



▪ Une mission RobotProg : le robot-peintre (2)

Objectif à atteindre :

Peindre l'entièreté du terrain : variation sur la mission précédente

Conditions initiales :

Terrain :	quelconque, sans obstacle
Position du robot :	quelconque
Direction du robot :	quelconque

Éléments du langage autorisés :

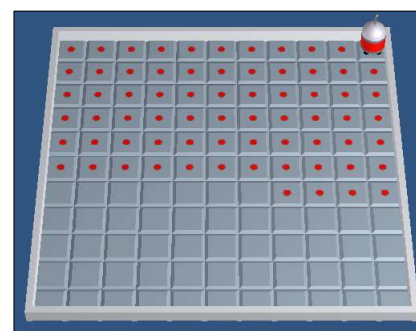
Procédures :	avancer, tournerGauche, tournerDroite, marquer
États logiques :	murEnFace, murAGauche, murADroite, caseMarquée
États numériques :	xRobot, yRobot, dxRobot, dyRobot
Instructions :	tantQue, Si Alors Sinon, affectation

Consignes particulières :

- Partir d'une case quelconque, peindre une 'moitié' de terrain depuis cette case, y revenir ensuite pour peindre l'autre 'moitié'

Variations possibles :

- aucune ---



▪ Une mission RobotProg : trajet minimum (3)

Objectif à atteindre :

Amener le robot sur une case particulière

Conditions initiales :

Terrain :	quelconque, sans obstacle
Position du robot :	quelconque
Direction du robot :	quelconque

Éléments du langage autorisés :

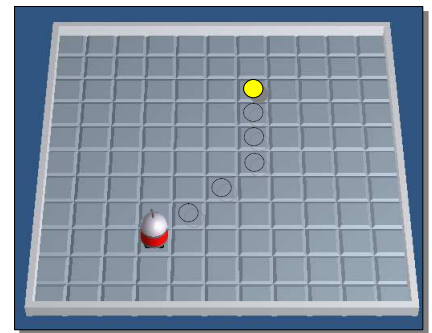
Procédures :	avancer, tournerGauche, tournerDroite
États logiques :	murEnFace, murAGauche, murADroite
États numériques :	distanceMur, dxRobot, dyRobot, xRobot, yRobot
Instructions :	tantQue, Si Alors Sinon, affectation

Consignes particulières :

- Atteindre l'objectif en un minimum de pas (du point de vue du spectateur)

Indications utiles :

- ci-contre ...



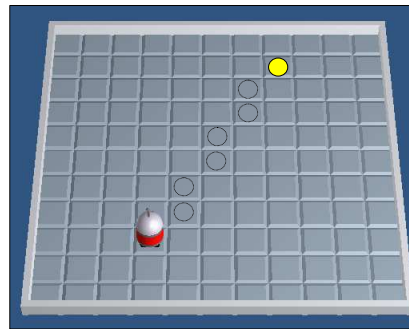
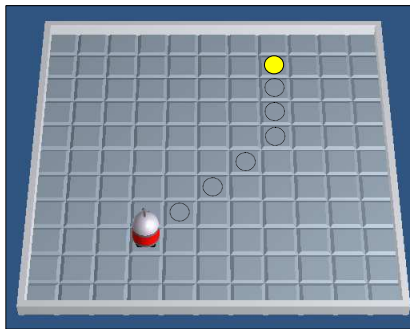
▪ Une mission RobotProg : trajet minimum (4)

Objectif à atteindre :

Amener le robot sur une case particulière, en parcourant une ligne 'la plus droite' possible entre le point départ et le point d'arrivée

on a fait ceci précédemment : bof ...

mieux (avec le même nombre de pas)



Conditions initiales :

Terrain :	quelconque, sans obstacle
Position du robot :	quelconque
Direction du robot :	quelconque

Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite
États logiques :	murEnFace, murAGauche, murADroite
États numériques :	distanceMur, dxRobot, dyRobot, xRobot, yRobot
Instructions :	tantQue, Si Alors Sinon, affectation

Consignes particulières :

- Atteindre l'objectif en un minimum de pas et au plus droit

Indications utiles :

- Intéressez-vous à l'algorithme de Bresenham :

ici : http://fr.wikipedia.org/wiki/Algorithme_de_tracé_de_segment_de_Bresenham

ou ici : <http://www.biometrie-online.net/dossiers/technique/commun/basiques.pdf>

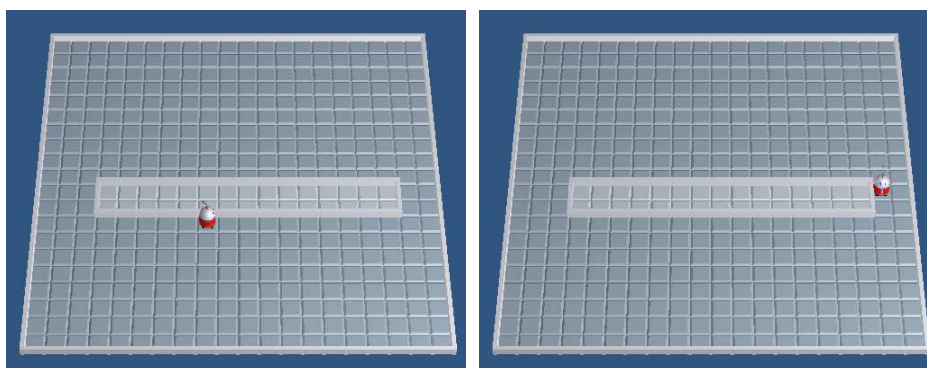
▪ Une mission RobotProg : longer un mur intérieur (1)

Objectif à atteindre :

Longer un mur droit et en faire une fois le tour complet

Conditions initiales :

Terrain :	quelconque, mur droit horizontal ou vertical - ne touchant pas les murs extérieurs (au moins une case vide) - d'épaisseur inconnue (min une case)
Position du robot :	quelque part contre le mur (horizontalement ou verticalement)
Direction du robot :	dans la direction du mur, mais sens inconnu



Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite
États logiques :	murEnFace, murAGauche, murADroite
États numériques :	distanceMur, dxRobot, dyRobot, xRobot, yRobot
Instructions :	tantQue, Si Alors Sinon, affectation

Consignes particulières :

- --- aucune ---

Indications utiles :

- --- aucune ---

Variantes :

- votre algorithme est-il toujours valable si le mur n'a pas d'épaisseur ?

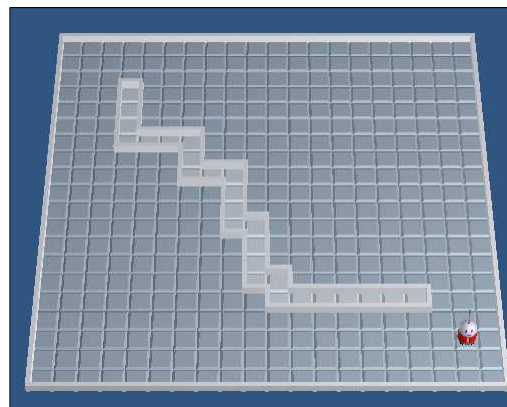
▪ Une mission RobotProg : longer un mur intérieur (2)

Objectif à atteindre :

Atteindre, puis longer un mur quelconque et en faire une fois le tour complet

Conditions initiales :

Terrain :	quelconque, mur 'en zigzag' - ne touchant pas les murs extérieurs (au moins une case vide) - d'épaisseur inconnue (au moins une case)
Position :	indéterminée
Direction :	indéterminée



Éléments du langage autorisés :

- --- mission précédente ---

Consignes particulières / Indications utiles :

- --- aucune ---

Variantes :

- quelle est la difficulté si le mur n'a pas d'épaisseur ?

▪ Une mission RobotProg : le robot-peintre (3)

Objectif à atteindre :

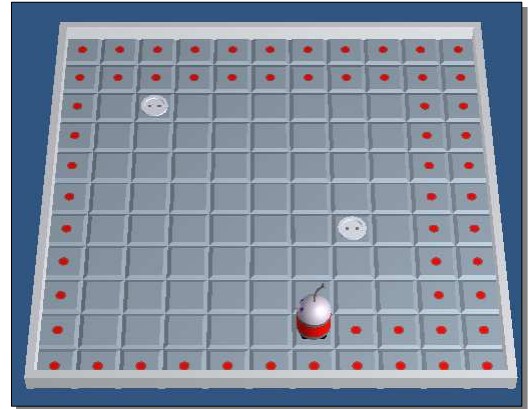
Peindre le terrain, en faisant le tour de l'extérieur vers l'intérieur

Conditions initiales :

Terrain :	quelconque, sans obstacle
Position du robot :	dans un coin
Direction du robot :	sens horlogique

Éléments du langage autorisés :

Procédures :	avancer, tournerGauche, tournerDroite, marquer, recharger
États logiques :	murEnFace, caseMarquée, robotSurPrise
États numériques :	--- aucun ---
Instructions :	tantQue, Si Alors Sinon



Consignes particulières :

- Si le terrain a une taille supérieure à 100 cases (10 x 10), comme l'algorithme est dévoreur d'énergie, il est utile de disposer çà et là des prises permettant la recharge des batteries

Indications utiles :

- --- aucune ---

Variantes :

- Rendre l'algorithme indépendant de la direction initiale
- Écrire une autre version avec la permission d'utiliser d'autres procédures ou états (logiques ou numériques) que ceux indiquées

▪ Une mission RobotProg : labyrinthe

Objectif à atteindre :

Trouver la prise, s'y recharger et s'arrêter

Conditions initiales :

Terrain :	comme illustré, des couloirs d'une case de large ; une prise au fond d'un des couloirs
Position du robot :	dans le coin, comme illustré
Direction du robot :	comme illustré

Éléments du langage autorisés :

- tous

Indications utiles :

- petit Poucet

Variantes :

- changer la prise de place

