Automates à états

I. Introduction

La théorie des automates a été essentiellement développée pour construire un modèle mathématique correspondant au fonctionnement des calculateurs, afin de répondre à la question fondamentale de tout programmeur : "Que peut faire ou ne pas faire un calculateur ?"
Déjà dans les années 30, Alan Turing se penchait sur le problème. La célèbre machine de Turing résulte de ses réflexions (il s'agit à la base d'un automate).

La science des automates - dispositifs assurant un enchaînement automatique et continu d'opérations arithmétiques et logiques (Larousse) - les répartit en plusieurs catégories :
. les automates à états finis,
. les automates à états finis déterministes et non déterministes,
. les automates à pile, déterministes et non déterministes,
. les machines de Turing,
. les automates à registres,
. ...

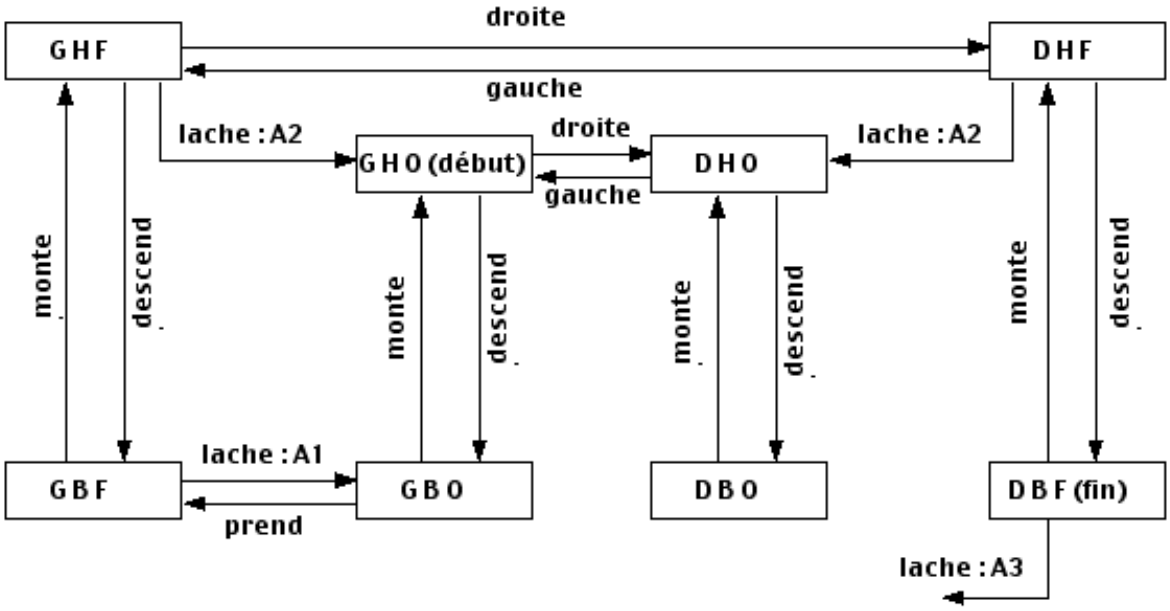
L'automate est une machine qui, à un instant donné, se trouve dans un état défini. Cet état évolue dans le temps suivant les événements qui surviennent.
A chaque événement une action peut être associée. Sur un événement, défini pour l'état en cours, l'action associée est exécutée puis l'état suivant devient l'état courant.

Ce concept peut être représenté de façon graphique, les états sont représentés par des boites et les événements par des liaisons fléchées entre les états:

Le schéma représente l'état "i" qui est activé par l'événement "y", sans action associée. L'état "i" accepte deux événements : "x" et "z" dont ce dernier possède une action associée.

L'assemblage de plusieurs états, dont les états "début" et "fin", définit la structure de l'automate.

Exemple du bras de robot :
(voir explications plus bas)



Le concept d'automate est particulièrement adapté à l'industrie de production avec notamment les chaînes d'assemblages, où chaque poste de montage est un automate.

Cependant, ce concept s'applique à de nombreux autres domaines :
. orgues de Barbaries,
. recettes de cuisine,
. jeux de rôles,
. machines à laver,
. feux de signalisation,
. langages de programmation (ce qui nous intéresse ici),
. ...

Nous n'exposerons ici que le principe des automates à états finis déterministes : AEFD. La formalisation d'un AEFD est une fonction allant d'un ensemble fini d'états K et d'un ensemble fini d'événements T vers l'ensemble fini d'états K : f: K x T -> K, avec k0 état initial de K.
L'automate est dit déterministe si la fonction f est injective, c'est à dire qu'un état donné ne peut pas accepter deux fois le même événement pour aller vers deux états différents.
En informatique on parle d'un langage "normal" si est seulement s'il s'inscrit dans un automate déterministe.

Les événements ou les entrées de l'automate forment une chaîne d'éléments, appelés les signes caractéristiques du langage.

II. Description sous forme de texte

L'automate se définit très simplement de façon graphique (voir exemple ci-dessus), mais peut se décrire aussi à l'aide d'un langage possédant seulement quelques éléments lexicaux. Il faut définir les mots clés désignant l'état considéré, les événements possibles et les actions associées.

Une transcription en langage naturel pourrait être :
à partir de l'état "i"
accepter l'événement "x"
pour prendre l'état "j"
accepter l'événement "y"
pour prendre l'état "k"
en exécutant l'action "a"
...
Comme notre propos n'est pas de proposer un programme d'interprétation du langage naturel, nous allons simplifier la construction du langage en le réduisant à quelques mots clés. Ceux-ci sont en anglais, l'anglais permet ici plus de concision, néanmoins pour les inconditionnels de la langue française la modification des mots clés ne pose aucun problème.

Les mots clés de base sont :
. from : pour définir l'état considéré
. event : pour spécifier l'événement à prendre en compte
. to : pour définir l'état suivant
. action `...` : pour décrire l'action à exécuter

Le texte de l'automate devient :

```
from i
  event x to j
  event y to k
    action `a`
```

Ce qui est nettement plus simple pour l'interpréteur.

III. Moteur de l'automate

L'exécution de l'automate se fait à l'aide d'un séquenceur. Le séquenceur se place dans le premier état de l'automate, l'état "début", et se met en attente d'un événement. Quand un événement attendu se produit, le séquenceur exécute l'action associée à cet événement (si elle existe), et se place dans l'état destination, pour se remettre en attente. Si l'événement n'est pas attendu par l'automate dans l'état courant, le séquenceur ne fait rien et se remet en attente.

On le remarque ici, la structure du noyau de l'automate est très simple. C'est un avantage majeur de

la programmation à l'aide d'automate. Le séquencement de l'automate est entièrement séparé des actions entreprises. On peut donc faire tourner l'automate sans exécuter d'action, pour vérifier le séquencement. Une fois le séquencement établit, on peut librement rajouter toutes les actions voulues et travailler avec, comme sur un programme normal.

La recherche de dysfonctionnements de l'automate est ainsi grandement facilitée.

IV. Exemple du robot

Voici un petit exemple d'automate s'appliquant à un bras de robot. Le bras peut se déplacer de haut en bas et de droite à gauche. Le bras comporte une pince, qui ne peut se fermer que sur un objet. Les états du robot sont "bras en bas à gauche pince ouverte" ou "bras en bas à droite pince fermée", etc. Les événements sont les ordres que lui donne l'opérateur : monte, descend, gauche, droite, prend, lâche. Avec ces ordres on peut réaliser tous les états. Ceux-ci constituent le langage du robot.
L'objectif de ce petit exemple est d'écrire l'automate correspondant, tout en sachant que les objets à prendre arrivent en bas à gauche du robot et que le robot ne peut aller de droite à gauche ou vice versa qu'une fois le bras en l'air. L'état initial est en haut à gauche pince ouverte et l'automate se termine quand le robot ouvre sa pince en bas à droite avec un objet à l'intérieur. Si le bras lâche l'objet en l'air, il est perdu et il faut recommencer.

Voilà donc le texte correspondant (fichier "RobSeq.txt") :

```
automate RobSeq  (TEvent, evNul, EventDes, TEventDes, GetNextEvent, `Ada.Text_IO, EventMgr`)
default
  event default
    to same
      action `Put_Line("Ordre incorrect !!!");`

from DroiteHautOuvert
  event Descend
    to DroiteBasOuvert
      action `Put_Line("Etat du bras : DroiteBasOuvert");`
  event Gauche
    to GaucheHautOuvert
      action `Put_Line("Etat du bras : GaucheHautOuvert");`

from GaucheHautOuvert
  event Descend
    to GaucheBasOuvert
      action `Put_Line("Etat du bras : GaucheBasOuvert");`
  event Droite
    to DroiteHautOuvert
```

```
    action `Put_Line("Etat du bras : DroiteHautOuvert");`

from DroiteBasOuvert
  event Monte
    to DroiteHautOuvert
      action `Put_Line("Etat du bras : DroiteHautOuvert");`

from GaucheBasOuvert
  event Prend
    to GaucheBasFerme
      action `Put_Line("Objet pris");`
  event Monte
    to GaucheHautOuvert
      action `Put_Line("Etat du bras : GaucheHautOuvert");`

from DroiteHautFerme
  event Lache
    to DroiteHautOuvert
      action `Put_Line("Objet perdu");`
  event Descend
    to DroiteBasFerme
      action `Put_Line("Etat du bras : DroiteBasFerme");`
  event Gauche
    to GaucheHautFerme
      action `Put_Line("Etat du bras : GaucheHautFerme");`

from GaucheHautFerme
  event Lache
    to GaucheHautOuvert
      action `Put_Line("Objet perdu");`
  event Descend
    to GaucheBasFerme
      action `Put_Line("Etat du bras : GaucheBasFerme");`
  event Droite
    to DroiteHautFerme
      action `Put_Line("Etat du bras : DroiteHautFerme");`

from DroiteBasFerme
  event Lache
    to out
      action `Put_Line("Objet posé. Fin.");`
  event Monte
    to DroiteHautFerme
      action `Put_Line("Etat du bras : DroiteHautFerme");`
```

```
from GaucheBasFerme
  event Lache
    to GaucheBasOuvert
      action `Put_Line("Objet lâché");`
  event Monte
    to GaucheHautFerme
      action `Put_Line("Etat du bras : GaucheHautFerme");`

end
```

(voir le graphe correspondant en introduction)

Une fois l'automate en exécution l'opérateur doit donner les bons ordres de façon à prendre un objet en bas à gauche et à le déplacer en bas à droite.
(voir génération de source Ada pour compiler le programme Robot)

Cet exemple démontre aussi la souplesse de la programmation par automate : en remplaçant les actions, qui sont ici des "Put_Line", par des commandes directes d'un bras de robot, on obtient immédiatement le programme de commande du robot.

V. Gestion des événements

Dans l'exemple précédent, on ne fait pas mention de la façon d'acquérir les événements. En effet, l'automate récupère les événements à travers le moteur. On voit ici un autre point fort de la programmation par automate : la gestion des événements est transparente pour l'automate, enfouie dans le moteur. Elle est donc uniforme pour tout les événements qui peuvent se produire : touches du clavier, souris, temporisation, message inter-tâches,
interruption matérielle,...

Le moteur fait appel à une primitive du style "GetNextEvent", pour acquérir le prochain événement. La construction de cette primitive est à la charge du programmeur. Celui-ci doit synthétiser tous les types d'événements possibles et leurs attribuer à chacun un identificateur ayant le type énuméré "TEvent". Le contexte de l'événement (code de la touche, valeur de la temporisation, ...) peut être passé par le paramètre "EventDes" de type "TEventDes".

La syntaxe de déclaration est :

```
procedure GetNextEvent(Event : out TEvent; EventDes : out TEventDes) is
begin
-- Attribution d'un identificateur à chaque événement pouvant survenir.
end;
```

(Voir un exemple de codage dans le listing du Robot : EventMgr.adb).

VI. Génération de sources Ada

Pour exécuter l'automate je propose un programme qui n'est ni vraiment un interpréteur, ni vraiment un compilateur, mais qui va coder le texte de l'automate en langage Ada, pour pouvoir ensuite le compiler avec son compilateur préféré.
Le texte de l'automate est mis sous la forme de package : "package <Nom de l'automate>". Cette unité doit être incluse dans le programme principal qui y fait référence en appelant la fonction "Start<Nom de l'automate>", avec comme paramètre : l'événement initial (souvent il s'agit de l'événement nul).
L'automate s'exécute alors jusqu'à l'état "Fin" où il rend l'exécution au programme principal, avec la valeur de la fonction à "True" dans le cas où il n'y aurait pas d'erreur, ou à "False" dans le cas où l'automate aurait pris l'état "Erreur".
L'unité <Nom de l'automate> fait appel à une unité "EventMgr" (ou plusieurs) qui contient la définition des événements, la gestion des événements "GetNextEvent" et toutes les références (types, variables, procédures,...) qui sont mentionnées dans le texte des actions.
Pour obtenir l'unité Ada correspondant au texte de l'automate, il faut exécuter "GenAuto". Le programme demande alors les éléments suivants :
- le type énuméré qui contient les événements (Exemple : "TEvent = (evNul, evX, evY, evZ);")
- le contexte de l'événement (Exemple : si "evX" est un événement de réception d'un message, le contexte sera l'accès au message)
- le type du contexte de l'événement (dans l'exemple précédent il s'agit d'un type accès)
- la constante représentant l'événement nul, c'est le seul événement imposé par le séquenceur. Il appartient au type énuméré des événements
- la procédure de gestion des événements, c'est elle qui établit la correspondance entre les événements extérieurs et les constantes décrites dans le type énuméré des événements
- la ou les unités à inclure, elles contiennent toutes les références utiles à l'unité générée (dans le cas de plusieurs unités, il faut séparer leur identificateur par des virgules : "EventMgr1, EventMgr2, EventMgr3"). L'unité "Ada.Text_IO" est automatiquement incluse.
- le nom du fichier contenant le texte de l'automate
- le nom des fichiers du package Ada généré .ads et .adb.

Prenons l'exemple du programme "Robot". Pour le rendre exécutable, il faut d'abord exécuter "GenAuto" avec les paramètres suivants : (Inclus dans le fichier de description robseq.auto)

. type des événements :	TEvent
. contexte de l'événement :	EventDes
. type du contexte de l'événement	TEventDes
. constante événement nul :	evNul
. procédure de gestion des événements :	GetNextEvent
. unité(s) à inclure :	EventMgr
. nom du fichier source :	RobSeq.txt
. nom de l'unité Ada :	RobSeq.ads / adb

ensuite compiler Robot.adb, RobSeq.adb et EventMgr.adb pour obtenir l'exécutable.

Voici les listings :

. EventMgr.ads / adb : unité de définition des types de bases, "TEvent" notamment et de la procédure "GetNextEvent" qui récupère une touche du clavier pour la transformer en un événement.

. RobSeq.ads / adb : unité contenant l'automate généré par "GenAuto" avec le texte source de l'automate "RobSeq.txt".

. Robot.adb : programme principal qui appelle l'automate.

VII. Fonctionnalités du langage de description d'automate

Le programmeur définit un automate avec le nombre d'états et d'événements qu'il souhaite, le nombre est juste limité par le compilateur Ada utilisé.
Un état se définit par son identificateur, par une partie "initialisation" puis une partie "description des événements".
La partie "initialisation" décrit les actions à exécuter avant de traiter les événements. La partie "description des événements" décrit chaque événement par son identificateur (qui peut être l'événement par défaut pour prendre en compte tous les événements, attention : les événements sont traités par ordre de description et donc le premier décrit sera le premier exécuté), l'état suivant et l'action déclenchée par l'événement.
L'état suivant peut être un état décrit par l'utilisateur avant ou après l'état en cours (l'ordre de définition importe peu, seul le premier état a une signification spéciale, car, c'est lui qui sera activé en premier), le même, l'état "Erreur" ou bien l'état de "Sortie" du séquenceur.
L'action est une instruction Ada ou un appel à un sous-automate. On peut appeler soit un état qui après une succession d'état reviendra à l'endroit de départ (du style du "gosub" en Basic), soit un automate externe qui est défini de la même façon que l'automate principal. Par exemple un automate de déplacement d'un robot fait appel au sous automate de déplacement de son bras pour déplacer une pièce, tel que celui définit plus haut.
Les parties "initialisation" et "événements" peuvent être définies au préalable pour tous les états en se rajoutant à celles définies localement, par contre l'événement par défaut ne sera effectif que s'il n'en existe pas localement, sinon celui définit localement à la primeur.
Une première détection des incohérences de syntaxe est réalisée au moment de la translation en package Ada. Les dernières incohérences seront détectées à la compilation du package, par exemple la correspondance entre la définition des états et leur utilisation.

VIII. Description du langage de définition de l'automate

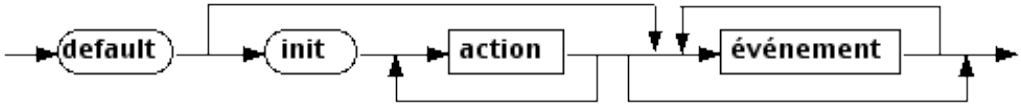
En plus des mots clés définis plus haut, d'autres existent. Les voici au complet :

. action `<code Ada>` :	défini le code Ada à exécuter double `` pour insérer un `
. automate <nom> :	défini le début du texte et le nom de l'automate
. call <nom> :	action particulière qui active un sous-automate externe, sans chaînage des événements
. debug	
. default :	. en début de texte, défini une action ou un événement à prendre par défaut . après event, défini tous les événements autres que ceux référencé après "event"
. end :	défini la fin du texte
. error :	défini un état particulier qui est l'état de sortie avec erreur
. event <evt>, <evt>..<evt>, default :	défini l'événement à prendre en compte
. from <état> :	défini le comportement de l'automate dans l'état considéré
. gosub <état> :	action particulière qui active un sous-automate, avec chaînage des événements
. init :	défini une action à faire quand l'automate prend l'état où init est défini
. out :	défini un état particulier qui est l'état de sortie de l'automate, ou du sous automate
. same :	défini un état particulier qui est l'état courant
. to <état> :	défini l'état suivant quand l'événement associé surgit
. {<commentaire>} :	défini une zone de commentaires ajout -- ... et (* ... *)
. + :	indique que l'événement courant est conservé pour la suite (chaînage dans le cas de sous-automates)

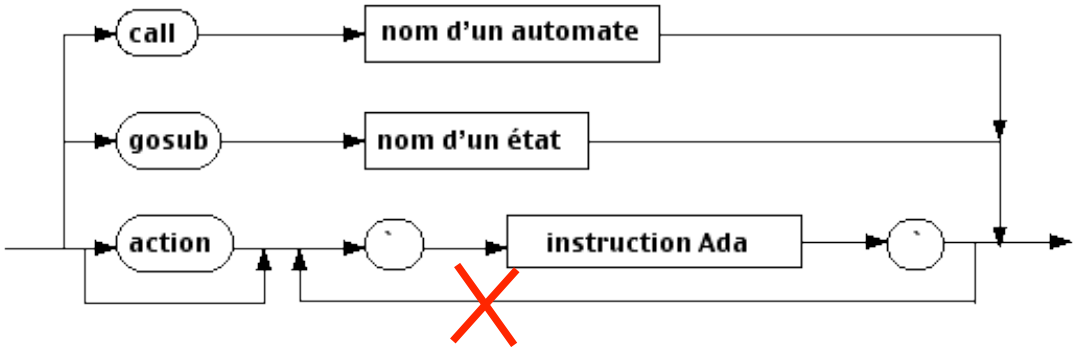
Voici la définition syntaxique :

Définition d'un automate :

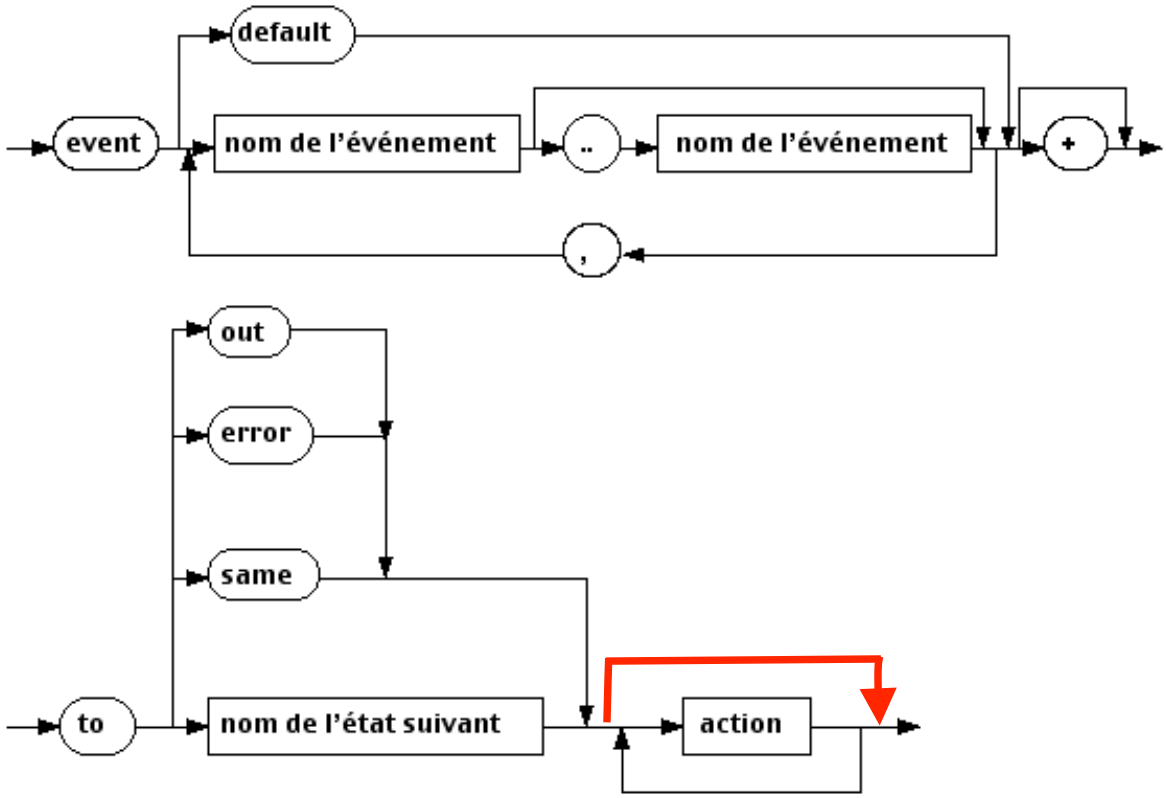
Définition des actions par défaut :



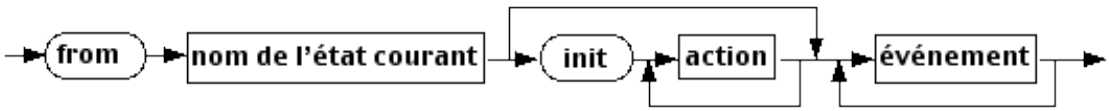
Définition d'une action :



Définition d'un événement :



Définition d'un état :



Note : les noms des automates, des états et des événements sont des identificateurs qui se soumettent aux même règles que les identificateurs Ada :

. ils commencent par une lettre

. ils comportent des lettres, des chiffres et le signe '_'

. les majuscules et les minuscules ne sont pas différenciées.

IX. Description du programme

Le programme est décomposé en un programme principal et trois unités :

. GenAuto : programme principal
. OutSrc : unité d'interprétation du texte de l'automate
. InSrc : unité qui assure la lecture du fichier source et sa transformation en mots clés qui constituent les événements de l'automate interpréteur
. BasicDef : unité contenant les définitions communes
. ~~ArbMgr~~ : ~~unité contenant un objet de gestion de l'arbre binaire~~

On remarque la présence de "SrcSeq", cette unité a été entièrement générée à partir d'un automate avec le programme "GenAuto". Cependant, comme l'histoire de l'oeuf et de la poule, il faut compiler GenAuto et donc SrcSeq avant de pouvoir le générer de façon automatique...

X. Description des objets

1. "TsourceMgr"

"TSourceMgr" est un objet construit pour lire un fichier texte, ici il s'agit du texte source de l'automate.
L'objet propose une procédure "Open" pour initialiser les champs de l'objet, ouvrir et lire le contenu du fichier dont le nom est passé en paramètre; une procédure "Read" pour lire le caractère courant et le suivant - s'il existe, du fichier texte; une procédure "Status" pour renvoyer le nom du fichier courant et le numéro de la ligne courante; une procédure "Close" pour libérer la mémoire.

~~2. "ArbMgr"~~

~~"ArbMgr" est un objet construit pour manipuler des chaînes de caractères triées par ordre alphabétique, au moyen d'une structure en arbre binaire, ici les mots clés de l'automate sont stockés dans l'arbre binaire(voir détails au chapitre "Arbre binaire").~~
~~L'objet propose une procédure "Ajoute" pour ajouter un élément dans l'arbre : le mot clé est ainsi associé à son identificateur Ada ; une procédure "Balance" pour optimiser la construction de l'arbre et donc minimiser le temps d'accès aux éléments; une procédure "Recherche" pour rechercher un élément dans l'arbre et retourner son identificateur Ada, si l'on n'a rien trouvé on renvoie "NonDefini"; une procédure "RetournePremier" qui renvoie le premier élément de l'arbre suivant la clé de tri; une procédure "RetourneSuivant" qui renvoie l'élément suivant suite à un précédent appel de "RetournePremier" ou "RetourneSuivant" une procédure "Detruit" pour libérer la mémoire.~~

3. "TTextListMgr"

"TTextListMgr" est un objet construit pour manipuler des chaînes de caractères au moyen d'une structure de listes chaînées. Ici les lignes du texte objet, le texte de l'unité Ada générée, sont stockées en listes. L'objet propose une procédure "Init" pour initialiser les champs de l'objet; une procédure "Add" pour ajouter une chaîne de caractères à la chaîne courante; une procédure "AddNew" pour clore la chaîne courante et créer une nouvelle chaîne avec celle passée en

paramètre; une procédure "WriteToFile" pour écrire les chaînes dans un fichier texte; une procédure "CopyTo" pour copier les chaînes dans une autre référence de liste de même type; une procédure "Done" pour libérer la mémoire.

4. "TEnumListMgr"
"TEnumListMgr" est un descendant de "TTextListMgr", ici les noms (évènements ou automates externes) sont stockés dans la liste chaînée.
L'objet propose en plus une procédure "AddUniq" pour vérifier l'existence du nom avant de l'ajouter à la liste chaînée; une procédure "TWriteToFile" pour écrire les noms sous forme d'une liste énumérée.

5. "TStateListMgr"
"TStateListMgr" est un descendant de "TEnumListMgr", ici les noms des états sont stockés dans la liste chaînée.
L'objet propose en plus une procédure "AWriteToFile" pour écrire les noms des états sous forme de noms de procédures; une procédure "CWriteToFile" pour écrire le nom du premier état comme paramètre de lancement de l'automate (le premier état de l'automate est activé en premier).

6. "ReadToken"
"ReadToken" est une procédure qui permet la lecture du fichier source de l'automate au moyen de l'objet "TSourceMgr" et le transforme en éléments lexicaux, compréhensibles par l'automate de génération de l'unité Ada.

XI. Arbre binaire (N'est plus utilsé)

Minimiser le temps de recherche d'un identificateur parmi ceux du langage est la principale raison d'emploi d'une structure d'arbre binaire pour mémoriser les identificateurs du langage.
L'arbre binaire est une structure de données liée à une relation d'ordre procurant un nombre moyen de comparaison pour la recherche d'un élément de l'ordre de log 2 n. C'est à dire que l'on effectue de l'ordre de 15 comparaisons pour trouver un élément parmi 32768 (2^15). Bien sur, dans le meilleur des cas, on effectue une seule comparaison si l'élément est le premier de l'arbre, et malheureusement 32768 si l'arbre est complètement déséquilibré (voir figure 1 et 2).

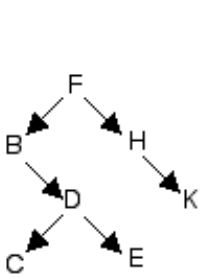


Figure 1

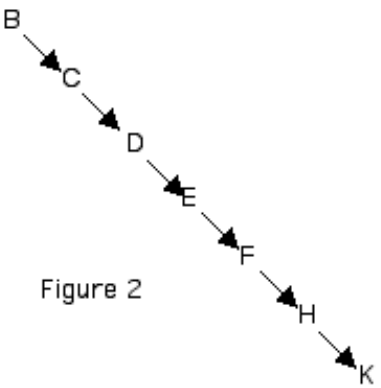


Figure 2

Pour l'arbre de la figure 1, il faut 3 comparaisons pour trouver l'élément "K" alors qu'il en faut 7 pour l'arbre de la figure 2. Voila, le mot est lâché : la performance d'un arbre binaire est liée à sa construction équilibrée ou non.
L'ordre d'ajout des éléments est donc très important lors de la construction de l'arbre.
Dans la plupart des cas on ne connaît pas à priori la valeur des éléments par avance, alors on doit se contenter du facteur "de l'ordre de log2 n" ou mettre en place des algorithmes de rééquilibrage de l'arbre ce que l'on verra ensuite.
Par contre, dans le cas qui nous intéresse, nous avons l'avantage de connaître tous les identificateurs du langage. Nous allons donc les ordonner correctement pour obtenir un arbre le plus équilibré qui soit.
L'algorithme de classement se résume à prendre à chaque fois l'élément central d'une suite d'éléments ordonnés, par dichotomie.

Pour nos identificateurs cela donne :
"action automate call default end error event from gosub init out to same".
L'élément central est "event".

Puis on constitue deux sous-listes :
"action automate call default end error" et "from gosub init out to same", dont les éléments centraux sont "call" et "init". (On remarque que si le nombre d'éléments est pair le choix de l'une ou l'autre possibilité n'affecte pas la performance finale, bien que les arbres soient différents).

Puis on constitue les sous-listes :
"action automate", "default end error", "from gosub", "out to same", dont les éléments centraux sont "action", "end", "from", "to".

Puis on constitue les sous-listes : "automate", "default error", "gosub", "out same", dont les éléments centraux sont "automate", "default", "gosub", "out".

Puis on constitue les sous-listes :
"error", "same", dont les éléments centraux sont "error", "same".

Ce qui donne de l'ordre de construction suivant :
"event call init action end from to automate default gosub out error same".
Et l'arbre binaire suivant :

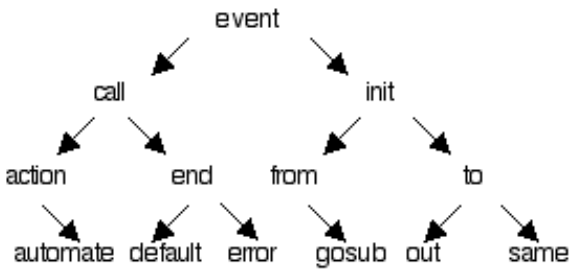


Figure 3

L'arbre est bien équilibré, le nombre de comparaisons maximum pour une recherche sera de 4 soit log 2 de 16.
Le codage de la construction ci-dessus peut s'obtenir en appelant la procédure "Ajoute" dans l'ordre indiqué. Cependant, ce n'est pas toujours possible ou alors cette contrainte est trop forte si la liste n'est pas figée.
Pour équilibrer notre arbre nous possédons une procédure dans l'objet que nous nommons "Balance". On peut appeler "Balance" chaque fois qu'il est nécessaire. C'est à dire, bien souvent soit après une série d'ajout d'éléments soit avant une recherche.
Le codage de "Balance" suit la démarche décrite précédemment en plaçant en premier lieu les éléments de l'arbre suivant la relation d'ordre puis en les replaçant dans l'arbre en suivant la dichotomie.
L'objet de gestion de l'arbre binaire devient assez important pour que l'on puisse le placer de façon autonome dans une unité de compilation nommée : "ArbMgr".
(source de l'unité arbmgr.ads/adb)
Cette unité est un package générique. Elle demande à définir le type de la clef de tri, le type de l'élément manipulé, la constante de l'élément non défini, l'autorisation d'appel automatique à l'équilibrage de l'arbre et le comparateur compatible du type de la clef.

XII. Historique du compilateur d'automate

Comme l'histoire de l'œuf et de la poule, l'unité qui interprète et génère l'unité Ada n'a pas été issue dans sa forme première du programme pour la bonne raison qu'il n'existait pas encore.
La démarche a été donc de bâtir en premier lieu le principe de base de l'automate l'état, avec ses évènements et ses actions. L'option d'un interpréteur graphique était séduisante, mais d'une part le portage d'une solution graphique est aléatoire suivant les systèmes et d'autre part la gestion du graphisme demande un effort non négligeable qui était non compatible avec l'obtention du programme final. Une grammaire basée sur un texte était de loin la plus adaptée au problème. Elle fut conçue autour de terme anglais, dont ce n'est pas l'apologie, mais l'anglais permet d'avoir des termes plus concis. Une translation avec des termes français est toujours possible. Il restait à définir la structure du moteur de l'automate au sein de l'unité Ada. La réalisation de la première unité Ada pour l'interprète a servi de définition. L'unité a été écrite manuellement comme le programme le ferait par la suite. Ensuite l'interpréteur s'est auto-amélioré au fur à mesure de l'évolution de l'automate de l'interpréteur.
Le programme a d'abord fait l'objet d'une version en langage Pascal pour être transcrite et améliorée en Ada.

XIII. Les listings

~~{ inclure le listing de ArbMgr.ads / ads }~~
{ inclure le listing de BasicDef.ads / adb }
{ inclure le listing de InSrc.ads / adb }
{ inclure le listing de OutSrc.ads / adb }
{ inclure le listing de GenAuto.adb }

XIV. Génération de l'unité "SrcSeq.ads / adb"

Pour obtenir "SrcSeq.ads / adb", il faut exécuter "GenAuto" avec les paramètres suivant :(Inclus dans srcseq.auto)

. type des événements :	TTokenId
. contexte de l'événement :	Token
. type du contexte de l'événement :	TTokenStr
. constante événement nul :	NullId
. procédure de gestion des événements :	ReadToken
. unité(s) à inclure :	InSrc, OutSrc, BasicDef, UXStrings, UXStrings.Text_IO
~~. nom du fichier source :~~	~~SrcSeq.txt~~
. nom de l'unité Ada :	~~SrcSeq.ads / .adb~~

lecture

Le programme interprète ensuite le texte de l'automate "SrcSeq.~~txt~~".
auto

XV. Améliorations possibles

Renforcer le contrôle de la syntaxe, par exemple dans l'automate proposé, rien n'empêche de définir l'événement "default" avant les autres.
Utilisation de la représentation graphique de l'automate avec traduction en expression textuelle.
Construire les éléments lexicaux à l'aide d'un automate.
Interpréter les paramètres depuis la ligne de commande.
Récupérer et traiter les exceptions.
Faciliter le portage sur d'autres plateformes.

XVI. Conclusions

Voici une idée de départ pour explorer le monde des automates et des langages.

Pascal Pignard, mai 2001.octobre 2023.