Zürich, 2. 2. 2005 / 15. 6. 2005

Les bonnes idées, de l'autre côté du miroir

Niklaus Wirth

Résumé

Tout un pot-pourri d'idées est répertorié, provenant des dernières décennies de la science et de la technologie informatiques. Largement acclamées à leur époque, beaucoup d'entre elles ont perdu en splendeur et éclat sous l'examen critique d'aujourd'hui. Nous essayons d’en trouver des raisons. Quelques-unes de ces idées sont presque oubliées. Mais nous pensons qu'elles méritent d'être rappelées, non seulement parce qu'on doit essayer de tirer parti des leçons du passé, mais aussi dans l'intérêt du progrès, de la stimulation intellectuelle ou du plaisir.

Table des matières

1. Introduction....2
2. La technologie du matériel....2
3. L'architecture des ordinateurs....3
3.1. La représentation des nombres....3
3.2. L'adressage des données....4
3.3. Les piles d'expression....6
3.4. Les données de longueur variable....6
3.5. La mémorisation des adresses de retour dans le code....7
3.6. L'adressage virtuel....7
3.7. La protection de la mémoire et les classes d'utilisateurs....8
3.8. Les jeux d'instructions complexes....9
3.9. Les fenêtres de registres....11
4. Les caractéristiques des langages de programmation....11
4.1. Les notations et la syntaxe....12
4.2. L’instruction GO TO....13
4.3. Les branchements multiples....14
4.4. L'instruction compliquée de boucle d'Algol....15
4.5. Les variables locales rémanentes ‘own’....15
4.6. Le passage de paramètres par nom d'Algol....16
4.7. La spécification incomplète des paramètres....17
4.8. Les failles de sécurité....18
5. D'autres techniques....19
5.1. L'analyse syntaxique....19
5.2. Langages extensibles....20
5.3. Procédures imbriquées et table de pointeurs de Dijkstra....20
5.4. Les tables des symboles à structure arborescente....22
5.5. L'usage de mauvais outils....23
5.6. Les assistants....24
6. Les paradigmes de programmation....24
6.1. La programmation fonctionnelle....24
6.2. La programmation logique....25
6.3. La programmation orientée objet....26
7. Remarques finales....26

1. Introduction

L'histoire de l'informatique a été marquée par de nombreuses idées fructueuses et originales. Mais quelques-unes d'entre elles se sont avérées moins géniales qu'elles ne le paraissaient au départ. Dans de nombreux cas, et c'est typique pour la technologie, leur importance a été réduite par les changements dans l'environnement technologique. Souvent, des facteurs commerciaux ont également influencé l'importance d'un bonne idée. Et certaines idées se sont simplement révélées moins efficaces et moins glorieuses, lorsqu'elles ont été examinées rétrospectivement, ou jugées après une analyse critique. D'autres idées se sont avérées être la réincarnation d'idées inventées antérieurement puis oubliées, peut-être parce qu'elles étaient en avance sur leur temps, peut-être parce qu'elles n'avaient pas flatté les modes et les tendances du moment. Et certaines idées ont été réinventées, bien qu'elles aient déjà été mauvaises dès l'origine.

Cela m'a conduit à l'idée de rassembler un certain nombre de ces bonnes idées qui se sont révélées être moins géniales rétrospectivement. La motivation pour le faire a été déclenchée par un récent entretien avec Charles Thacker au sujet des *idées obsolètes* - les idées se détériorant avec l'âge. J'ai également redécouvert un article de Don Knuth intitulé « *The dangers of computer-science theory* » (Les dangers de la théorie de l'informatique). Le discours de Thacker a été prononcé derrière la Muraille de Chine, celui de Knuth en Roumanie en 1970 derrière le Rideau de fer, deux lieux sûrs contre la néfaste « *critique occidentale* ». Le document de Knuth en particulier, avec son attitude de langue de bois, m'a encouragé à écrire ce recueil d'histoires. Celui-ci ne prétend pas être exhaustif.

Ce recueil commence par quelques concepts infructueux dans le domaine de la technologie informatique et se poursuit ensuite avec des idées contestables en matière d'architecture des ordinateurs. On peut justifier certaines d'entre elles dans le contexte de leur époque, mais elles sont à présent obsolètes. Vient ensuite un ensemble d'idées issu du domaine des langages de programmation, en particulier des premiers protagonistes. Enfin, nous abordons divers sujets à propos des paradigmes de la programmation liés à la technologie du logiciel (compilateur).

2. La technologie du matériel

La vitesse a toujours été la préoccupation dominante et presque exclusive des ingénieurs informaticiens. Le perfectionnement des techniques existantes a été l'une des voies empruntées pour atteindre cet objectif, en recherchant des solutions de rechange. Certaines de ces recherches n'ont pas abouti, et voici quelques-unes d'entre elles.

À l'époque où les mémoires à tore magnétique étaient dominantes, l'idée d'une *mémoire à bulles magnétique* est apparue. Comme d'habitude, on y attachait de grands espoirs. On prévoyait de remplacer toutes sortes de dispositifs à rotation mécanique, qui étaient les principales sources de problèmes et de manque de fiabilité. Bien que les bulles magnétiques tournaient toujours sous un champ magnétique dans un matériau de ferrite, il n'y aurait aucune pièce mécaniquement mobile. Comme les disques, c'était un dispositif séquentiel . Mais elles n'ont jamais atteint une capacité suffisante, et les progrès en technologie des disques étaient tels que non seulement la capacité, mais aussi la vitesse des bulles devenaient inférieure. L'idée a été discrètement enterrée après quelques années de recherche.

Une nouvelle technologie qui a entretenu de grands espoirs pendant des décennies est celle des *appareils cryogéniques*, en particulier dans le domaine des superordinateurs. On promettait des

vitesses de commutation ultra élevées, mais l'effort pour faire fonctionner les grands équipements informatiques à une température proche du zéro absolu était prohibitif. L'apparition d'ordinateurs personnels utilisables sur votre table laissa les rêves cryogéniques s'évaporer ou se figer.

Ensuite, il y a eu l'idée d'utiliser des *diodes tunnel* à la place des transistors comme éléments de commutation et de mémoire. La diode tunnel (ainsi nommée parce qu'elle reposait sur un effet quantique des électrons passant par-dessus une barrière énergétique sans avoir l'énergie nécessaire) a une caractéristique particulière avec un segment négatif. Cela permet de coder deux états stables. La diode tunnel est un dispositif au germanium et n'a pas d'équivalent à base de silicium. Cela limita son fonctionnement sur une plage de température relativement étroite. Les transistors au silicium sont devenus plus rapides et moins chers à un rythme tel que les chercheurs ont pu oublier la diode tunnel.

Le même progrès phénoménal dans la fabrication des transistors au silicium n'a jamais permis aux transistors à l’arséniure de gallium de répondre aux attentes élevées pour lesquelles ils avaient été introduits. Le silicium possède un avantage intrinsèque tenant au fait que son propre oxyde est un isolant idéal, ce qui simplifie les procédés de fabrication. D'autres matériaux tels que l'arséniure de gallium - la technologie 3-5 en général - sont aujourd'hui rarement utilisés en technologie informatique, mais ont leur application dans la niche de communication à très haute fréquence.

Il semble même maintenant que, cette fois, les jours du transistor bipolaire prépondérant sont comptés, du fait que les transistors à effet de champ sont toujours plus rapides, plus petits et moins chers, car il est devenu possible de parvenir à des couches d'oxyde extrêmement fines (gates). Toutefois, cela ne signifie pas que le transistor bipolaire ait été une mauvaise idée.

3. L'architecture des ordinateurs

3.1. La représentation des nombres

L'architecture des ordinateurs est un domaine où il est gratifiant de trouver de « bonnes » idées. La question fondamentale à l'époque était le choix de la représentation des nombres, en particulier des entiers. La question clé était le choix de leur base. Pratiquement tous les premiers ordinateurs, à l'exception de celui de Konrad Zuse, utilisaient la base 10, c'est-à-dire une représentation par des chiffres décimaux, comme tout le monde en a l'habitude et l'a appris à l'école.

Cependant, il est clair qu'une représentation binaire avec des chiffres binaires est beaucoup plus économique. Un entier n nécessite $log_{10}(n)$ chiffres décimaux en décimal et $log_2(n)$ chiffres binaires (bits) en représentation binaire. Comme un chiffre décimal nécessite 4 bits, la représentation décimale nécessite environ 20% de mémoire de plus que le binaire, ce qui montre l'avantage évident de la forme binaire. Pourtant, la représentation décimale a été conservée pendant longtemps, et même persiste aujourd'hui sous la forme de modules de bibliothèque.

La raison en est que les gens ont insisté pour croire que tous les calculs devaient être exacts. Cependant, des erreurs se produisent en raison des arrondis, par exemple après la division. Les effets de l'arrondi peuvent varier en fonction de la représentation des nombres, et un ordinateur binaire peut donner des résultats différents de ceux d'un ordinateur décimal. Parce que traditionnellement les transactions financières - et c'est là que l'exactitude compte ! - ont été calculées à la main avec des arithmétiques décimales, il a été estimé que les ordinateurs devaient, en tout état de cause, produire les mêmes résultats, en d'autres termes, commettre les mêmes erreurs.

Bien que la forme binaire donne en général des résultats plus précis, la forme décimale est restée la forme préférée dans les applications financières, car un résultat décimal peut facilement être vérifié à la main si nécessaire. Bien que cela soit peut-être compréhensible, il s'agissait

clairement d'une idée conservatrice. Il convient de mentionner que jusqu'à l'avènement du système IBM 360 en 1964, qui gérait à la fois les arithmétiques binaire et décimale, les fabricants de grands ordinateurs ont conservé deux lignes de produits, à savoir les ordinateurs binaires pour leurs utilisateurs scientifiques et des ordinateurs décimaux pour leurs clients commerciaux. Une pratique coûteuse !

Dans les premiers ordinateurs, les entiers étaient représentés par leur grandeur et un bit de signe séparé. Dans les machines qui reposaient sur l'addition séquentielle chiffre par chiffre, le signe était placé en tête afin d'être lu en premier. Lorsque le traitement parallèle des bits est devenu possible, le signe a été placé en dernier, toujours par analogie avec la notation couramment utilisée sur le papier. Cependant, l'utilisation d'une représentation en signe et grandeur était une mauvaise idée, car l'addition nécessite alors des circuits différents pour les nombres positifs et négatifs. Représenter les nombres négatifs par leur complément était manifestement une solution de loin supérieure, car l'addition et la soustraction pouvaient maintenant être effectuées par le même circuit. Certains concepteurs ont choisi des compléments à 1, où $-n$ est obtenu à partir de n en inversant simplement tous les bits, certains ont choisi des compléments à 2, où $-n$ est obtenu en inversant tous les bits et en ajoutant ensuite 1. Le premier a l'inconvénient de présenter deux formes pour le zéro (0...0 et 1...1). C'est fâcheux, en particulier si les instructions de comparaison disponibles sont inadéquates. Par exemple, dans l'ordinateur CDC 6000, il existait une instruction testant le zéro, reconnaissant correctement les deux formes, mais une instruction testant le bit de signe uniquement, classant 1...1 comme un nombre négatif, faisant des comparaisons inutilement compliquées. Il s'agissait d'un cas de conception inadéquate, révélant que le complément à 1 est une mauvaise idée. Aujourd'hui, tous les ordinateurs utilisent l'arithmétique du complément à 2.

en décimal	en complément à 2	en complément à 1
2	010	010
1	001	001
0	000	000 ou 111
-1	111	110
-2	110	101

Les nombres à parties fractionnaires peuvent être représentés en format à virgule fixe ou flottante. Aujourd'hui, le matériel informatique est généralement doté d'une arithmétique à virgule flottante, c'est-à-dire une représentation d'un nombre x par deux entiers, un exposant e et une mantisse m, tels que $x = B^e \times m$. Pendant un certain temps, il y a eu une discussion sur le choix de la base de l'exposant B. Le Burroughs B5000 a introduit B = 8, et l'IBM 360 a utilisé B = 16, tous deux en 1964, contrairement au conventionnel B = 2. L'intention était de gagner de la place grâce à un intervalle d'exposants plus petit, et pour accélérer la normalisation, car les décalages se produisent par groupes de 3 (ou 4) bits. Mais cela s'est avéré être une mauvaise idée, car cela a aggravé les effets d'arrondi. En conséquence, il a été possible de trouver les valeurs x et y sur l'IBM 360, telles que, pour certains petits, positifs ε, $(x+ \varepsilon)\times(y+ \varepsilon) < (x \times y)$. La multiplication avait perdu sa monotonie ! Une telle multiplication est peu fiable et potentiellement dangereuse.

3.2. L'adressage des données

Les instructions des premiers ordinateurs consistaient simplement en un code d'opération et une adresse absolue (ou une valeur littérale) en champ paramètre. Cela a rendu l'auto-modification par le programme inévitable. Par exemple, si des nombres stockés dans des cellules de mémoire consécutives devaient être ajoutés dans une boucle, l'adresse de l'instruction d'ajout devait être modifiée à chaque étape en y ajoutant 1. Bien que la possibilité de modifier le programme au

moment de l'exécution ait été annoncée comme l'une des grandes conséquences de l'idée profonde de John von Neumann de stocker le programme et les données dans la même mémoire, il s'est rapidement avéré que cela autorisait une technique dangereuse et constituait une source illimitée de pièges. Le code du programme doit rester inchangé, sinon la recherche d'erreurs peut devenir un cauchemar. On a reconnu la modification du code par lui-même comme une très mauvaise idée.

La solution consistait à introduire un autre mode d'adressage qui permettait de traiter une adresse comme un élément de donnée modifiable plutôt que comme une instruction du programme (ou une partie de celle-ci), qu'il vaut mieux ne pas toucher. La solution était le mode d'*adressage indirect* et la seule modification de l'adresse directement pointée (un mot de données). Bien que cela ait supprimé le danger de modification des programmes par eux-mêmes, et bien qu'elle soit restée une caractéristique commune de la plupart des ordinateurs jusqu'au milieu des années 1970, il faut la considérer rétrospectivement comme une idée discutable. Après tout, il fallait deux accès à la mémoire pour chaque accès aux données, et cela induisait donc un ralentissement considérable du calcul.

La situation s'est aggravée par l'idée « intelligente » d'une *indirection à plusieurs niveaux*. Les données accédées indiquaient par un bit si le mot référencé était la donnée souhaitée, ou s'il s'agissait à nouveau d'une autre adresse (peut-être encore une fois elle-même indirecte). Ces machines (par exemple HP 2116) ont été facilement mises au rancart en spécifiant une boucle d'adresses indirectes.

La solution réside dans l'introduction de registres d'index. À la constante d'adresse dans l'instruction serait ajoutée la valeur stockée dans un registre d'index. Pour cela, il fallait ajouter quelques registres d'index (et un additionneur) à l'accumulateur de l'unité arithmétique. L'IBM 360 les a tous regroupés en un seul banc de registres, comme c'est maintenant la coutume.

Une option particulière a été utilisée par les ordinateurs CDC 6000 : les instructions ne se référaient directement qu'aux registres, ceux-ci étant organisés en 3 bancs : registres de données de 60 bits (X), registres d'adresse de 18 bits (A) et les registres d'index 18 bits (B). L'accès à la mémoire était implicitement provoqué par toute référence à un registre A (dont la valeur avait été modifiée en ajoutant la valeur d'un registre B). Ce qui est étrange, c'est que les références à A0 - A5 impliquent la recherche de l'emplacement mémoire adressé dans le registre X0 - X5 correspondant, alors que la référence à A6 ou A7 implique le stockage de X6 ou X7. Bien que cet arrangement n'ait pas posé de gros problèmes, il est rétrospectivement juste de le classer comme une idée médiocre, car un numéro de registre détermine l'opération effectuée, c'est-à-dire le sens du transfert des données. En dehors de cela, le CDC 6000 présentait plusieurs excellentes idées, principalement sa simplicité. Il peut véritablement être appelée la première machine RISC, bien qu'elle ait été conçue par S. Cray en 1962, bien avant que ce terme ne soit inventé vers 1980.

On a inventé un mode d'adressage beaucoup plus sophistiqué avec la machine Burroughs B5000. On fait référence à son schéma de descripteurs, utilisé principalement pour désigner les tableaux. Un *descripteur de données* était essentiellement une adresse indirecte, mais il contenait en outre des bornes permettant de vérifier l'index au moment de l'accès. Bien que la vérification automatique de l'index soit excellente et presque visionnaire, l'introduction des descripteurs était une idée discutable, car les matrices (tableaux multidimensionnels) nécessitent un descripteur d'un tableau de tels descripteurs, un pour chaque ligne (ou colonne) de la matrice. Chaque accès à une matrice de dimension *n* nécessitait *n* indirections. De toute évidence, ce système n'a pas seulement ralenti les accès en raison des indirections, mais a également nécessité des rangées supplémentaires de descripteurs. Néanmoins, cette mauvaise idée a été adoptée par les concepteurs de **Java** en 1995 et de **C#** en 2000.

3.3. Les piles d'expression

Le langage **Algol 60** n'a pas seulement eu une profonde influence sur le développement des langages de programmation, mais - dans une mesure beaucoup plus limitée - également sur l'architecture des ordinateurs. Cela ne devrait pas être surprenant, car le langage, le compilateur et l'ordinateur forment un tout inextricable.

Le premier sujet à mentionner à cet égard est l'évaluation des expressions qui, en **Algol**, pouvait être d'une complexité quelconque, les sous-expressions mises entre parenthèses et les opérateurs ayant leurs propres forces de liaison[1]. Les résultats des sous-expressions doivent être sauvegardés temporairement. A titre d'exemple, prenons l'expression :

(a/b) + ((c+d)*(c-d))

On l'évaluerait selon les étapes suivantes, ce qui donnerait des résultats temporaires *t1 ... t4* :

t1 := a/b ; t2 := c+d ; t3 := c-d ; t4 := t2*t3 ; x := t1 + t4

F. L. Bauer et E. W. Dijkstra ont proposé indépendamment un schéma d'évaluation d'expressions quelconques. Ils ont remarqué qu'en évaluant de gauche à droite, en obéissant aux règles de priorité et parenthèses, le dernier élément résultat est toujours le premier à être nécessaire. Il pouvait donc être commodément placé dans une liste déroulante (une pile) :

t1 := a/b ; t2 := c+d ; t3 := c-d ; t2 := t2*t3 ; x := t1 + t2

Une idée immédiate consistait à réaliser cette stratégie simple en utilisant un bloc de registres, avec l'ajout d'un compteur incrémenteur/décrémenteur pour pointer le registre du sommet. Une telle pile réduisait le nombre d'accès à la mémoire et évitait la référence explicite des registres individuels dans les instructions. En bref, les ordinateurs à piles semblaient être une excellente idée. La technique a été mise en œuvre par l'entreprise anglaise Electric KDF-9 et les ordinateurs Burroughs B-5000. Cela a évidemment ajouté à la complexité de leur matériel.

Quelle doit être la profondeur d'une telle pile ? Après tout, les registres étaient des ressources coûteuses. Le B-5000 a choisi d'utiliser 2 registres seulement, et un vidage mémoire automatique par empilement si plus de deux résultats intermédiaires devaient être mémorisés. Cela semblait raisonnable. Comme Knuth l'avait souligné dans une analyse de nombreux programmes **Fortran**, l'écrasante majorité des expressions ne nécessitent qu'un ou deux registres. Néanmoins, l'idée d'une pile d'expressions s'est avérée être plutôt discutable, surtout après l'avènement des architectures avec des bancs de registres au milieu des années 1960. Aujourd'hui, la simplicité de la compilation a été sacrifiée pour tout gain de vitesse d'exécution. L'organisation de la pile avait réduit l'utilisation d'une ressource rare à une stratégie fixe. Mais maintenant, des algorithmes de compilation sophistiqués ont prouvé qu'ils utilisaient les registres de façon plus économique, étant donné la flexibilité de spécifier des registres individuels dans chaque instruction.

3.4. Les données de longueur variable

Les ordinateurs orientés vers le marché du traitement des données se caractérisaient généralement par une arithmétique décimale, et aussi par des capacités de gestion de données de longueur variable. Les textes (chaînes de caractères) et les nombres (chiffres décimaux) prédominent dans ces applications. L'IBM 1401 a été clairement orientée vers le traitement des chaînes, et il a été fabriqué en très grandes quantités. Il n'avait pas une longueur de mot de 8 bits - la taille de caractère prédominante à cette époque était de 6 bits ! - mais un octet de 9 bits (le mot *octet* n'était pas encore courant non plus avant 1964). L'un des bits dans chaque mot servait à marquer la fin de la chaîne ou de nombre, et on l'appelait le bit d'arrêt *stop-bit*. Les instructions pour

1 On parle habituellement de priorité entre opérateurs.

déplacer, ajouter, comparer et translater traitaient séquentiellement octet par octet et se terminaient en rencontrant un bit d'arrêt.

Cette solution de traitement des données de longueur variable semble être une assez mauvaise idée. Après tout, elle « gaspille » 11 % de mémoire consacrée aux bits d'arrêt, qu'ils soient nécessaires ou pas. Plus tard, les ordinateurs ont résolu ce problème par logiciel sans support matériel. Généralement, soit la fin d'une chaîne de caractères est marquée par un octet nul, ou le premier octet indique la longueur. Là aussi, les informations relatives à la longueur doivent être mémorisées, mais uniquement pour les chaînes de caractères, et non pour chaque octet en mémoire.

3.5. La mémorisation des adresses de retour dans le code

L'instruction de saut de sous-programme, inventée par D. Wheeler, mémorise le compteur ordinal dans un emplacement d'où il est récupéré à la fin du sous-programme. La question est : « Où se trouve cet emplacement ? » Dans plusieurs ordinateurs, notamment les mini-ordinateurs, mais aussi l'ordinateur d'entreprise[2] CDC 6000, une instruction de saut vers l'emplacement d déposait l'adresse de retour en *d*, puis poursuivrait l'exécution à l'emplacement *d+1* :

mem[d] := PC+1 ; PC := d+1

C'était certainement une mauvaise idée pour au moins deux raisons. Premièrement, elle empêchait qu'un sous-programme puisse être appelé de façon récursive. La récursivité a été introduite par **Algol** et a suscité beaucoup de controverses, parce que les appels de procédure, comme les appels de sous-programme, ne pouvaient plus être traités de cette manière simple, car un appel récursif écrasait l'adresse de retour de l'appel précédent. L'adresse de retour a donc dû être récupérée dans un endroit fixe dicté par le matériel et recopiée dans un endroit spécifique à l'invocation particulière de la procédure récursive. Ce « coût de supervision » semblait inacceptable pour beaucoup de concepteurs d'ordinateurs ainsi qu'aux utilisateurs, et ils en sont donc venus à la déclarer indésirable, inutile et interdite. Ils n'ont pas voulu remarquer que la difficulté était due à leur instruction d'appel inadéquate.

La deuxième raison pour laquelle cette solution était une mauvaise idée est qu'elle empêchait le multitraitement. Chaque processus concurrent devait utiliser sa propre copie du code. C'est le simple résultat de l'erreur consistant à ne pas séparer le code du programme et les données. Nous laissons de côté la question de ce qui se passerait si une interruption se produisait après la sauvegarde du compteur ordinal PC et avant son affectation.

Certaines conceptions matérielles ultérieures, notamment les architectures RISC des années 1990, ont traité les appels de procédure récursifs en introduisant des registres spécifiques dédiés pour sauvegarder par empilement les adresses de retour. Nous reviendrons ultérieurement sur ce sujet, mais mentionnons simplement ici que la sauvegarde de l'adresse de retour dans l'un des registres à usage général (en supposant qu'il existe un banc de registres) est probablement la meilleure idée, car elle laisse la liberté de choix au concepteur du compilateur tout en gardant l'instruction de base des sous-programmes aussi efficace que possible.

3.6. L'adressage virtuel

Tout comme les concepteurs de compilateur avaient demandé aux architectes du matériel de fournir des fonctionnalités répondant à leurs besoins, les concepteurs de systèmes d'exploitation ont mis en avant leurs idées favorites. De telles demandes sont apparues avec l'avènement du multitraitement et du temps partagé, concepts qui ont donné naissance aux systèmes d'exploitation en général. L'idée directrice était d'utiliser le processeur de manière optimale, en le basculant vers un autre programme dès que celui en cours d'exécution était bloqué, par exemple, par une opération

2 Traduction de mainframe.

d'entrée/sortie. Les différents programmes ont ainsi été exécutés de façon entrelacée, quasi simultanément. En conséquence, les demandes d'allocation (et de libération) de mémoire se sont produites selon des séquences imprévisibles et arbitraires. Néanmoins, on a compilé chaque programme en supposant un adressage linéaire, un bloc contigu de mémoire. Pire encore, la mémoire physique n'était généralement pas assez grande pour s'adapter à un nombre suffisant de processus afin de rendre le multitraitement avantageux.

La solution intelligente à ce dilemme a été trouvée avec l'adressage indirect, cette fois-ci caché au programmeur. La mémoire était subdivisée en blocs ou en pages de longueur fixe (une puissance de 2). Une adresse (virtuelle) était convertie en une adresse physique en utilisant une table des pages. En conséquence, les pages individuelles pouvaient être placées n'importe où en mémoire et, bien que dispersées, apparaissaient comme une zone contiguë. Mieux encore, les pages ne trouvant pas leur emplacement dans la mémoire principale pourraient être vidées en mémoire secondaire, sur des disques de grande capacité. Un bit dans l'entrée de la table de page correspondante indiquait si les données étaient actuellement sur disque ou en mémoire principale.

Ce mécanisme intelligent et complexe a été utile en son temps. Mais il a montré ses difficultés et problèmes. Il fallait pratiquement que tout matériel moderne soit équipé de tables de pages et d'unités de conversion d'adresse, et cache le coût de l'adressage indirect - sans parler du vidage et de la restauration des pages sur le disque à des instants imprévisibles - à l'utilisateur peu méfiant. Même aujourd'hui, la plupart des processeurs utilisent le mécanisme de pagination[3], et la plupart des systèmes d'exploitation fonctionnent dans les environnements multi-utilisateur. Mais aujourd'hui, l'idée est devenue discutable, car les mémoires à semi-conducteurs sont devenues si grandes que l'astuce de la cartographie et du vidage n'est plus bénéfique. Néanmoins, le temps de supervision de l'adressage indirect et son mécanisme complexe restent présents.

Ironiquement, le mécanisme de l'adressage virtuel continue à être utilisé dans un but pour lequel il n'avait jamais été prévu : le déroutement des références à des objets non existants, la détection de l'utilisation des pointeurs NIL. On représente NIL par 0, et la page à l'adresse 0 n'est jamais attribuée. Cette utilisation abusive du coûteux système d'adressage virtuel est un sale truc, et ce problème aurait dû être résolu de manière plus immédiate.

3.7. La protection de la mémoire et les classes d'utilisateurs

Il y a un concept qui, à première vue, peut encore justifier le mécanisme de cartographie : la *protection*. Tout système permettant plusieurs utilisateurs simultanés doit prévoir une protection contre les interférences mutuelles. On doit garantir la protection mutuelle du programme et des données d'un utilisateur vis-à-vis d'autres utilisateurs. Une telle situation exige, de façon intrinsèque, une distinction entre les utilisateurs ayant des droits différents, appelés privilèges. En principe, deux classes suffisent, une pour les programmes d'utilisateurs « ordinaires » soumis à des restrictions d'accès, et l'autre pour un programme « superviseur » avec des droits d'accès illimités afin de pouvoir d'une part, allouer et affecter des blocs de mémoire aux nouveaux programmes utilisateur et d'autre part, les recycler après leur fin d’exécution. Si l’un de ces programmes utilisateur tentait d'accéder à la mémoire en dehors de la zone qui lui était allouée, probablement en raison d'une erreur, il était alors dérouté et le contrôle passait au programme de supervision qui était, supposé, exempt d'erreurs. Les droits d'accès étaient facilement enregistrés dans des tables de pages.

En y regardant de plus près, on se rend compte que le besoin de protection et des classes de programmes découle du fait que les programmes sont peut-être erronés dans la mesure où ils émettent des demandes d'accès mémoire en dehors de l'espace qui leur est alloué, ou tentent d'accéder à des dispositifs qu'ils ne devraient pas directement manipulés. Si tous les programmes

3 Aujourd'hui, deux niveaux d'indirection sont souvent utilisés, en l'occurrence la segmentation et la pagination.

étaient écrits dans un langage de programmation approprié, cela ne devrait même pas être possible, car - si le langage est correctement mis en œuvre – on ne pourrait pas référencer des ressources non mentionnées dans le programme. La protection doit être implicite dans les programmes compilés ; elle devrait être garantie par le logiciel.

Il faut rappeler aux lecteurs qui doutent de la légitimité de ce principe, que les systèmes de programmation s'appuient sur un ramasse-miettes automatique pour la gestion mémoire. Un ramasse-miettes a besoin d'une sécurité d'accès identique à celle d'un système multi-utilisateurs. Sans cette sécurité, un ramasse-miettes peut détruire « accidentellement » toute information, même dans un système à utilisateur unique à tout moment. Un système véritablement sûr devrait exiger que tout soit produit par des compilateurs corrects, et que ces derniers et tout le code généré soient non modifiables. On constate que cette exigence amène à renoncer presque totalement à la glorieuse idée de von Neumann constatant que les programmes sont accessibles sous forme de données dans leur mémoire commune.

C'est pourquoi, la protection matérielle apparaît comme une béquille, ne couvrant qu'une petite partie des transgressions possibles, et dont on peut se passer si les logiciels sont mis en œuvre en toute sécurité. Il apparaît maintenant comme ayant été une bonne idée à l'époque qui aurait dû être remplacée entre-temps.

3.8. Les jeux d'instructions complexes

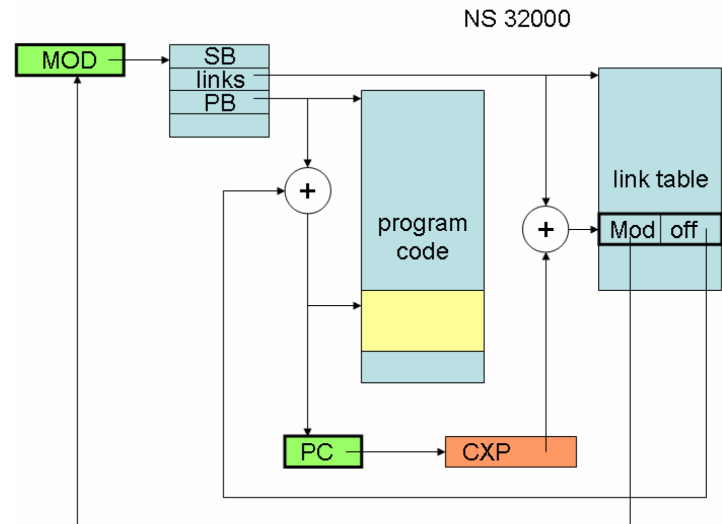
Les premiers ordinateurs comportaient de petits ensembles d'instructions simples, car ils devaient fonctionner avec un minimum de circuits coûteux. Le matériel devenant moins cher, la tentation s'est accrue d'intégrer des instructions de nature plus compliquée, telles que des sauts conditionnels avec trois cibles, des instructions qui incrémentent, comparent et sous condition effectuent un branchement de façon atomique, ou des opérations complexes de déplacement et de conversion. Avec l'avènement des langages de haut niveau, le désir s'est fait jour d'accorder certains concepts de ces langages avec des instructions adaptées en conséquence. Un bon exemple est celui du **for** d'**Algol** ou des instructions pour les appels de procédure (récursifs). Des instructions adaptées à ces mécanismes étaient une idée intelligente, car elles contribuaient à la densité du code, ce qui était important à une époque où la mémoire était une ressource rare, constituée de 64 K octets ou moins.

Cette tendance s'était manifestée dès 1963. On a adapté la machine Burroughs B5000 non seulement à de nombreuses caractéristiques complexes d'**Algol** - nous en reparlerons plus tard - mais elle réunissait à la fois un ordinateur scientifique et une machine de gestion de chaînes de caractères, elle regroupait deux ordinateurs avec des jeux d'instructions différents. Une telle extravagance était devenue possible avec la technique des microprogrammes stockés en mémoire morte rapide. Cette fonctionnalité a rendu aussi possible l'idée d'une famille d'ordinateurs : la série des IBM 360 se composait d'un ensemble d'ordinateurs, tous avec le même jeu d'instructions et la même architecture, du moins en ce qui concernait la partie visible au programmeur. Cependant, en interne, les machines étaient très différentes. En bas de gamme, les machines étaient microprogrammées, la version matérielle exécutant un microprogramme court pour l'interprétation du code d'instruction. Les machines haut de gamme, cependant, exécutaient toutes les instructions directement. Cette technologie s'est poursuivie avec des microprocesseurs mono-puces comme les Intel 8086, Motorola 68000 et National 32000.

Le processeur NS est un bon exemple comportant un jeu d'instructions complexes (CISC). Le regroupement de modèles d'instructions fréquentes en une seule instruction a permis d'améliorer la densité du code et a réduit le nombre d'accès à la mémoire, ce qui a augmenté la vitesse d'exécution.

Le processeur NS s'est adapté, par exemple, au nouveau concept de module et de compilation séparée avec une instruction d'appel appropriée. Les segments de code ont été édités au chargement, le compilateur ayant fourni des tables avec des informations de liaison. Il s'agit certainement d'une bonne idée pour minimiser le nombre d'opérations de liaison, qui remplacent des références aux tables de liens par des adresses absolues. Ce schéma, qui simplifie la tâche de l'éditeur de liens, amène à l'organisation mémoire suivante pour chaque module :

Un registre dédié MOD indique le descripteur du module M, qui contient la procédure P en cours d'exécution. Le registre PC est le compteur ordinal normal. Le registre SB contient l'adresse du segment de données de M, qui contient les variables statiques globales. Tous ces registres modifient leurs valeurs, chaque fois qu'une procédure externe est appelée. Pour accélérer ce processus, le processeur propose le CXP (procédure d'appel externe) en plus du BSR régulier (branchement sur sous-programme). Bien entendu, une paire d'instructions de retour correspondantes est disponible : RXP, RTS.



Disposition d’un module en mémoire

Supposons maintenant qu'une procédure P dans un module M doive être activée. Le paramètre d de l’instruction CXP spécifie l'entrée dans la table de liens actuelle. On obtient ainsi l'adresse du descripteur de M, et aussi le décalage de P dans le segment de code de M. Depuis le descripteur, ensuite, l'adresse du segment de données de M est obtenue et chargée dans SB. Tout cela avec une seule et courte instruction ! Cependant, ce qui a été gagné en simplicité de liens et en densité de code doit être payé quelque part, notamment par un nombre accru de références indirectes et, accessoirement, par du matériel supplémentaire, les registres MOD et SB.

Un deuxième exemple similaire est celui d'une instruction de vérification des limites d'un tableau. Elle compare la valeur d’un index aux limites inférieure et supérieure du tableau, et provoque un déroutement si l'index sort des limites, combinant ainsi deux instructions de comparaison et de branchement en une seule.

Plusieurs années après que notre compilateur **Oberon** ait été construit et diffusé, de nouvelles versions plus rapides du processeur sont apparues. Elles ont suivi la tendance à mettre en œuvre des concepts directement au niveau du matériel, et de laisser les instructions complexes être interprétées par un micro-code interne. En conséquence, ces instructions orientées vers le langage sont devenues plutôt lentes par rapport aux opérations simples. J'ai donc décidé de programmer une nouvelle version du compilateur qui s'est abstenu d'utiliser les instructions sophistiquées. Le résultat a été étonnant ! Le nouveau code était considérablement plus rapide que l'ancien. Il semble qu'à la fois, l'architecte du processeur et nous même, en tant que concepteurs de compilateurs, nous avions tous les deux « optimisé » au mauvais endroit.

En effet, au début des années 80, les microprocesseurs avancés ont commencé à concurrencer les anciens qui comportaient des jeux d'instructions très complexes et irréguliers. En fait, les jeux d'instructions étaient devenus si complexes que la plupart des programmeurs ne pouvaient en utiliser qu'une petite partie. De même, les compilateurs n’exploitaient qu’un sous-ensemble des instructions disponibles, un signe clair que les architectes du matériel avaient dépassé les bornes. La réaction s'est manifestée sous la forme des ordinateurs à jeu d'instructions réduit

(RISC), notamment les architectures Arm, Mips et Sparc des années 1990. Elles comportaient un petit ensemble d'instructions simples, toutes s'exécutant en un seul cycle d'horloge, un seul mode d'adressage et un banc unique assez important de registres, en bref : une structure très régulière. Ils ont démystifié les CISC en montrant que c'était une mauvaise idée !

3.9. Les fenêtres de registres

Lorsqu'une procédure est appelée, un bloc de mémoire doit être alloué pour ses variables locales. Lorsque la procédure est terminée, cet espace doit être libéré. Pour ce faire, la pile est le mécanisme le plus pratique, car la libération mémoire est gratuite et n'implique que la réinitialisation d'une adresse (pointeur de pile) dans un registre.

Une technique souvent utilisée pour l'optimisation du code consiste à allouer les variables les plus fréquemment utilisées dans des registres. Dans l'idéal, toutes les variables devraient résider dans des registres. Cependant, si seulement quelques variables dans chaque bloc de procédure restaient dans des registres, un gain de vitesse substantiel pourrait déjà être réalisé. C'est ainsi qu'est née l'idée des *fenêtres de registres* : sur appel d'une procédure, l'ensemble courant des registres est préservé, et un nouvel ensemble devient disponible pour le nouveau bloc. Comme il était prévisible que la mémoire deviendrait plus rapide, plus grande et moins chère dans l'avenir, de plus en plus de ces registres ont été placés en mémoire rapide sans que le programmeur ait à adapter ses programmes. C'était l'idée de l'*extensibilité*. Elle a conduit au *Processeur Sparc* avec le mécanisme intégré de fenêtres de registres : parmi la pile complète de registres, seuls ceux au sommet – c'est-à-dire dans la fenêtre - sont accessibles. Chaque appel de procédure ouvre une nouvelle fenêtre, chaque terminaison la dépile.

Cela semblait être une idée plutôt intelligente. Mais elle s'est avérée douteuse, bien que l'architecture Sparc a survécu comme l'un des exemples réussis de RISC's, et elle a été copiée plus tard par le processeur Itanium d'Intel. La raison pour laquelle elle doit être considérée comme douteuse, est qu'à tout moment seuls les registres actifs en sommet, ceux qui sont dans la fenêtre le plus récemment ouverte, sont directement utilisables et sont donc fréquemment accédés. Les autres sont quasi dormants, mais consomment une ressource effrayante et coûteuse. Ainsi, les fenêtres de registres impliquent une mauvaise utilisation de la ressource la plus chère.

4. Les caractéristiques des langages de programmation

Un terrain fertile pour les idées controversées est le sujet des langages de programmation. Ici, certaines des idées furent non seulement discutables, mais reconnues comme mauvaises dès le départ. Nous allons essayez d'éviter de discuter des mérites et des atrocités de chaque langage individuellement. Nous allons plutôt concentrer notre attention sur les concepts et constructions éventuellement introduits dans plusieurs langages. Nous prendrons pour base principalement les caractéristiques proposées par **Algol** en 1960 et par certains de ses successeurs [1].

Avant de commencer à énumérer les différentes caractéristiques, il semble nécessaire d'expliquer sur quelle base les évaluer. La plupart des gens considèrent un langage de programmation comme un simple code dans le seul objectif de construire des logiciels destinés à être « exécutés » par des ordinateurs. Nous considérons un langage comme un modèle de calcul et les programmes comme des textes formels se prêtant au raisonnement mathématique. Le modèle doit être défini de telle façon que sa sémantique soit définie sans référence à un mécanisme sous-jacent, qu'il soit physique ou abstrait. Il est évident que cela laisse apparaître un ensemble complexe de caractéristiques et d'installations, expliquées dans de gros volumes de manuels, comme une idée manifestement mauvaise. En fait, un langage n'est pas tant caractérisé par ce qu'il permet de programmer, mais surtout par ce qu'il empêche d'être énoncé. En citant le feu E. W. Dijkstra, la

tâche quotidienne la plus difficile du programmeur est de ne pas tout gâcher. Il me semble que le premier et noble devoir d'un langage est d'aider dans cette lutte éternelle.

4.1. Les notations et la syntaxe

Il est devenu à la mode de considérer la notation comme une question secondaire dépendant uniquement d'un goût personnel. Cela peut être en partie vrai ; cependant, le choix de la notation ne doit pas être considéré comme une question arbitraire. Elle a des conséquences, et elle révèle le caractère d'un langage.

Un exemple notoire d'une mauvaise idée est le choix du signe égal pour désigner l'affectation. Il remonte à **Fortran** en 1957 et a été copié à l'aveuglette par des armées des concepteurs de langage. Pourquoi est-ce une mauvaise idée ? Parce qu'elle bouleverse une tradition centenaire pour le signe « = » qui indique un test d'égalité, un prédicat qui est soit vrai soit faux. Mais **Fortran** l'a adopté pour signifier l'affectation, le *forçage* de l'égalité. Dans ce cas, les opérandes ne sont pas sur un pied d'égalité : l'opérande de gauche (une variable) doit être rendu égal à l'opérande de droite (une expression). x = y ne signifie pas la même chose que y = x. **Algol** a corrigé cette erreur par la solution simple : que l'affectation soit notée « := ».

Cela peut sembler pinailler aux programmeurs qui se sont habitués au signe égal qui signifie affectation. Mais mélanger affectation et comparaison est une très mauvaise idée, parce qu'elle exige qu'un autre symbole soit utilisé pour ce qui était traditionnellement exprimé par le signe égal. La comparaison pour l'égalité a été désignée par les deux caractères « == » (d'abord en **C**). C'est une conséquence de nature désagréable et cela a donné lieu à de mauvaises idées similaires en introduisant « ++ », « – », « && », etc.

Certains de ces opérateurs exercent des effets de bord (en **C**, **C++**, **Java** et **C#**), une source notoire d'erreurs de programmation. On pourrait accepter de laisser, par exemple, le signe ++ indiquer incrémentation de 1, si elle ne correspondait pas aussi à la valeur de l'incrément (ou la valeur devant être incrémentée ?), permettant ainsi des expressions avec des effets de bord. Le cœur du problème réside dans l'élimination de la distinction fondamentale entre *instruction* et *expression*. La première est une instruction, et elle est *exécutée* ; la seconde représente une valeur qui doit être calculée, et elle est *évaluée*.

La laideur d'une construction apparaît généralement en combinaison avec d'autres caractéristiques du langage. En **C**, on peut écrire, par exemple, x+++++y, une énigme plutôt qu'une expression, et un défi pour un analyseur sophistiqué ! Devinez quoi ? Sa valeur est-elle égale à ++x+++y+1 ? Ou bien la réponse suivante est-elle correcte ?

x+++++y+1 == ++x+++y   x+++y++ == x+++++y+1

On est tenté de postuler une nouvelle algèbre ! Il est en effet absolument surprenant de voir comment ce monstre de notation a été accepté par la communauté des programmateurs du monde entier.

Une rupture similaire avec les conventions établies a été le postulat que les opérateurs étaient associés par la droite dans le langage **APL** en 1962. *x+y+z* signifiait soudain *x+(y+z)*, et *x-y-z* s'interprétait *x-y+z*. Quel piège traître !

Un cas de syntaxe malheureuse plutôt qu'un simple mauvais choix de symbole a été celui d'**Algol** dans son instruction conditionnelle. Elle était proposée sous deux formes, S0, S1 étant les déclarations :

if b **then** S0
if b **then** S0 **else** S1

Cette définition a donné lieu à une ambiguïté inhérente et est devenue connue sous le nom de problème du **else** ‘pendouillant’ ... Par exemple, la déclaration :

if b0 **then if** b1 **then** S0 **else** S1

peut être interprétée de deux manières, à savoir :

if b0 **then** (**if** b1 **then** S0 **else** S1)
if b0 **then** (**if** b1 **then** S0) **else** S1

qui pourrait conduire à des résultats très différents. L'exemple suivant semble encore plus grave :

if b0 **then for** i := 1 **step** 1 **until** 100 **do if** b1 **then** S0 **else** S1

car il peut être analysé de deux façons, ce qui donne des calculs très différents :

if b0 **then** [**for** i := 1 **step** 1 **until** 100 **do if** b1 **then** S0 **else** S1]
if b0 **then** [**for** i := 1 **step** 1 **until** 100 **do if** b1 **then** S0] **else** S1

Le remède est cependant assez simple : Utiliser un symbole **end** de fin explicite dans chaque construction qui est récursive et commence par un symbole de départ explicite, comme **if, while, for, case**:

if b **then** S0 **end**
if b **then** S0 **else** S1 **end**

4.2. L’instruction GO TO

Quel autre concept pourrait mieux servir de point de départ à une liste de mauvaises idées ? L'instruction **goto** a été la cible de nombreuses critiques. Elle est le pendant direct en langages de l'instruction de branchement dans les jeux d'instructions machine. Elle peut être utilisée pour construire des instructions conditionnelles ou répétitives. Mais elle permet aussi de construire n'importe quel flot de contrôle labyrinthique ou déstructuré lors du déroulement des programmes. Elle défie toute structure régulière, et rend difficile un raisonnement structuré sur ces programmes, voire impossible.

Expliquons pourquoi l'instruction **goto** est devenue le prototype d'une mauvaise idée en langages de programmation. Les premiers outils dans notre lutte pour comprendre et contrôler les objets complexes sont la structure et l'abstraction. Un objet trop complexe est décomposé en parties. La spécification de chaque partie fait abstraction des aspects qui ne sont pas pertinents pour l’ensemble, dont la pertinence est locale par rapport à l'objet lui-même. C'est pourquoi, la conception d'ensemble peut se réaliser avec une connaissance limitée aux spécifications de l'objet, à son interface.

En corollaire, un langage doit permettre, encourager, voire imposer la formulation de programmes en tant que structures correctement imbriquées, dans lesquelles les propriétés de l'ensemble peuvent être dérivées à partir des propriétés des composants. Considérons, par exemple, la spécification d'une répétition R d'une déclaration S. Il s'ensuit que S apparaît comme une partie de R. Nous montrons deux formes possibles :

R0 : **while** b **do** S **end**
R1 : **repeat** S **until** b

La clé d'une imbrication correcte est que les propriétés connues du S peuvent être utilisées pour dériver les propriétés de **R**. Par exemple, étant donné qu'une condition (assertion) P est laissée valide (invariante) sous exécution de **S**, nous concluons que P est également laissé invariant lorsque l'exécution de **S** est répétée. Ceci est formellement exprimé par les règles de Hoare :

{P & b} **S** {P} implique {P} **R0** {P & ¬b}

{P} **S** {P}   implique {P} **R1** {P & b}

Si, toutefois, **S** contient une instruction **goto** , aucune assertion de ce type n'est possible à propos de **S**, et donc aucune déduction sur l'effet du **R**. C'est une grande perte. La pratique a en effet montré que les gros programmes sans **goto** sont beaucoup plus faciles à comprendre, et qu'il est beaucoup plus facile de donner des garanties sur leurs propriétés.

On en a assez dit et écrit sur ce point pour convaincre presque tout le monde que c'est un exemple de mauvaise idée. Le concepteur de **Pascal** a retenu l'instruction **goto** (ainsi que l'instruction **if** sans symbole **end** de fin). Apparemment, il a manqué de courage pour rompre avec les conventions et a fait de mauvaises concessions aux traditionalistes. Mais ça, c'était en 1968. Aujourd'hui, presque tout le monde a compris le problème, mais apparemment pas les concepteurs des derniers langages de programmation commerciaux, tels que le **C#**.

4.3. Les branchements multiples

Si un mécanisme est une mauvaise idée, alors les mécanismes construits par-dessus sont encore pires. Cette règle peut bien être démontrée par le concept de branchement multiple (switch, select, case, etc...). Un branchement multiple est essentiellement un tableau d'étiquettes.

En supposant, par exemple, que L1, ... L5 sont des étiquettes, une déclaration d'aiguillage en **Algol** peut ressembler à ce qui suit :

```
switch S := L1, L2, if x < 5 then L3 else L4, L5
```

Alors, la déclaration apparemment simple **goto** S[i] est équivalente à

```
if i = 1 then goto L1 else
if i = 2 then goto L2 else
if i = 3 then
        if x < 5 then goto L3 else goto L4 else
if i = 4 then goto L5
```

Si le **goto** est adapté à la programmation d'un désordre, le branchement multiple le rend inévitable.

C.A.R. Hoare a proposé en 1965 une solution de remplacement mieux adaptée : l'instruction **case**. Cette construction présente une structure appropriée avec des instructions composantes à sélectionner en fonction de la valeur i :

```
case i of
        1 : S1 | 2 : S2 | .............. | n : Sn
end
```

Cependant, les concepteurs de langages de programmation modernes ont choisi d'ignorer cette élégante solution en faveur d'une formulation qui est un bâtard entre le **switch Algol** et une instruction **case** structurée :

```
switch (i) {
        case 1 : S1 ; break;
        case 2 : S2 ; break;
        ... ;
        case n : Sn ; break;
}
```

Soit le symbole **break** indique une séparation entre des branches consécutives, soit il agit comme un **goto** à la fin de la construction du **switch**. Dans le premier cas, il est superflu, dans le

second cas, c'est un **goto** déguisé. Un mauvais concept dans une mauvaise notation ! L'exemple est tiré du C.

4.4. L'instruction compliquée de boucle d'Algol

Les concepteurs d'**Algol** ont reconnu que certains cas fréquents de répétition seraient mieux exprimés sous une forme plus concise qu'en combinant des instructions **goto**. Ils ont introduit l'instruction de boucle **for**, qui est particulièrement pratique dans le cadre des tableaux, comme par exemple dans :

for i := 1 **step** 1 **until** n **do** a[i] := 0

Si nous pardonnons le choix plutôt malheureux des mots *step* et *until*, cela semble une excellente idée. Malheureusement, la bonne idée a été infectée par une mauvaise idée, l'idée de généralité imaginative. La séquence de valeurs successives pris par la variable de contrôle *i* peut être spécifié sous forme de liste :

for i := 2, 3, 5, 7, 11 **do** a[i] := 0

En outre, ces éléments pouvaient être des expressions générales :

for i := x, x+1, y-5, x*(y+z) **do** a[i] := 0

Cela ne suffisant pas, il fallut aussi permettre différentes formes d'items de liste :

for i := x-3, x **step** 1 **until** y, y+7, z **while** z < 20 **do** a[i] := 0

Naturellement, les esprits intelligents concoctaient rapidement des pathologies, démontrant l'absurdité du concept :

for i := 1 **step** 1 **until** i **do** a[i] := 0
for i := 1 **step** i **until** i **do** i := - i

La généralité du **for** d'**Algol** aurait du être un signal d'alerte pour tous les futurs concepteurs de garder toujours en tête l'objectif premier d'une structure de contrôle et de se méfier de généralités abusives et complexes, qui deviennent rapidement contre-productives.

4.5. Les variables locales rémanentes 'own'

Algol avait introduit le concept de *localité*. Chaque procédure a sa propre portée, impliquant que les identifiants déclarés dans la procédure étaient locaux et invisibles à l'extérieur de la procédure. Il s'agit probablement de l'innovation la plus importante introduite par **Algol**. Pourtant, il s'est avéré qu'elle n'était pas tout à fait adaptée à certaines situations. En particulier, lors de l'entrée dans une procédure, toutes les variables locales sont des variables nouvelles. Cela implique qu'un nouvel espace mémoire doit être alloué à l'entrée et libéré à la sortie de la procédure. Jusqu'à présent, tout va bien. Cependant, dans certains cas, il peut être souhaitable de se souvenir de la valeur d'une variable à la sortie lorsque la procédure est exécutée une nouvelle fois. Évidemment, son espace mémoire pourrait alors ne pas être libéré. **Algol** a prévu cette option en permettant simplement qu'on déclare la variable comme **own**. Un exemple populaire pour ce cas est une procédure de génération de nombres pseudo-aléatoires :

real procedure random; **own real** x; **begin** x := (x*a + b) **mod** c; random := x **end**

Cet exemple simple montre déjà l'essentiel de ce concept : comment est initialisée la première valeur de la variable *x* appartenant à la procédure ? Toute solution s'avère être plutôt lourde. Une autre question non résolue est celle de la signification du terme « **own** » dans le cas de la récursion. De toute évidence, le problème est plus profond. La simple introduction du symbole **own** avait manifestement créé plus de problèmes qu'elle n'en avait résolus.

Le problème réside dans l'idée astucieuse d'**Algol** de lier le concept de portée des identificateurs avec celui de durée de vie des objets. Un identificateur devient visible lorsque l'objet identifié voit le jour (à l'entrée du bloc), et devient invisible lorsqu'il a cessé d'exister (à la sortie du bloc). La connexion étroite entre la visibilité et l'existence devait être rompue. Cela a été fait plus tard (dans **Mesa**, **Modula**, **Ada**) en introduisant le concept de *module* (ou de paquetage). Les variables d'un module sont globales, mais ne sont accessibles que par les procédures déclarées dans le module. Ainsi, lorsqu'une procédure (locale au module et exportée du module) est terminée, les variables du module continuent d'exister, et lorsque la procédure est rappelée, les anciennes valeurs réapparaissent. Le module masque les variables (masquage d'information), qui sont initialisées soit lors du chargement du module, soit par un appel explicite d'une procédure d'initialisation.

Cet exemple montre les problèmes qui apparaissent lorsque différents concepts – ici le masquage d'information et la vivacité - sont représentées par une seule construction linguistique. La solution consiste à utiliser des constructions distinctes pour exprimer des concepts différents, le module pour le masquage d'informations, la procédure pour le caractère dynamique des données. Un cas similaire de mariage malheureux est donné par certains langages orientés objet, qui lient entre eux (1) d'une part, des objets avec des pointeurs et (2) d'autre part, des modules avec des types, appelés classes.

4.6. Le passage de paramètres par nom d'Algol

Algol a introduit des procédures et des paramètres de manière beaucoup plus générale que dans des langages plus anciens (**Fortran**). En particulier, les paramètres ont été considérés comme dans les mathématiques fonctionnelles, où le paramètre réel remplace *textuellement* le paramètre formel [3]. Par exemple, étant donné la déclaration

real procedure square(x); **real** x; square := x * x

l'appel *square(a)* doit être interprété littéralement comme *a*a*, et *square(sin(a)*cos(b))* comme *sin(a)*cos(b) * sin(a)*cos(b)*. Cela nécessite l'évaluation du sinus et du cosinus à deux reprises, ce qui, selon toute vraisemblance, n'était pas l'intention du programmeur. Afin d'éviter ce cas fréquent et trompeur, un deuxième type de paramètre a été postulé en plus du paramètre nom ci-dessus : le paramètre valeur. Cela signifie qu'une variable locale (x') doit être allouée et initialisée avec la valeur du paramètre réel (x). Avec :

real procedure square(x); **value** x; **real** x; square := x * x

l'appel ci-dessus devait être interprété comme :

x' := sin(a) * cos(b) ; square := x' * x'

évitant ainsi la double évaluation inutile du paramètre réel. Le passage de paramètre par nom est vraiment un dispositif très souple, comme le montrent les exemples suivants.

Soit la déclaration :

```
real procedure sum(k, x, n);
integer k, n; real x;
begin real s; s := 0;
    for k := 1 step 1 until n do s := x + s;
    sum := s
end
```

Maintenant, la somme $a_1 + a_2 + ... + a_{100}$ s'écrit simplement comme *sum(i, a[i], 100)*, le produit interne des vecteurs *a* et *b* comme *sum(i, a[i]*b[i], 100)* et la fonction harmonique comme *sum(i, 1/i, n)*. Mais la généralité, aussi élégante et sophistiquée qu'elle puisse paraître, a son prix.

Un peu de réflexion révèle que chaque passage de paramètre par nom doit être implanté sous la forme d'une procédure anonyme sans paramètres. Le programmeur qui ne se doute de rien paiera-t-il volontiers le coût de supervision ? Bien sûr, un compilateur bien conçu peut « optimiser » certains cas. Le concepteur intelligent va se réjouir du défi que représente un si bel objet !

Le lecteur se demandera peut-être à ce stade : « Pourquoi toute cette histoire ? ». Peut-être ira-t-il plus loin en suggérant de supprimer simplement du langage le mode de passage de paramètres par « nom ». Toutefois, cette mesure serait trop drastique et donc inacceptable. Cela empêcherait par exemple l'affectation aux paramètres afin de transmettre des résultats à l'appelant. Cette suggestion a toutefois conduit au remplacement du mode de passage par nom par le mode de passage par référence dans des langages ultérieurs tels que **Algol W**, **Pascal**, **Ada**, etc. Aujourd'hui, le message est le suivant : Soyez sceptique à l'égard de caractéristiques et de mécanismes trop sophistiqués. Au minimum, leur coût pour l'utilisateur doit être connu avant qu'un langage ne soit délivré, publié et distribué. Ce coût doit être proportionnel aux avantages obtenus par l'élément.

4.7. La spécification incomplète des paramètres

Cette question est presque une suite de la précédente, car elle concerne les paramètres. Il se trouve que le langage n'exige pas de spécifications de type complètes pour les paramètres. Cela apparaît maintenant presque comme une négligence, mais avec de graves conséquences. **Algol** permettait, par exemple, la déclaration suivante :

real procedure f(x, y); f := (x-y)/(x+y)

Étant donné les variables :

integer k; **real** u, v, w

les appels suivants sont possibles :

u := f(2.7, 3.14) ; v := f(f(u+v), 10) ; w := f(k, 2)

Jusqu'à présent, tout va (plus ou moins) bien. Ce qui suit est plus problématique :

real procedure g; g := u*v + w;
u := f(g, g)

Apparemment, on ne peut pas déduire de la déclaration de *f*, si une variable, une expression ou une fonction sera substituée à un paramètre formel donné, ni si les paramètres de la fonction sont corrects en nombre et en types. Lorsqu'un tel paramètre formel est accédé, un test au moment de l'exécution doit d'abord déterminer si le paramètre effectif correspondant est une variable, une expression ou une fonction. Il s'agit d'un surcoût inacceptable dans la plupart des applications pratiques.

L'histoire pourrait s'arrêter là, n'ayant qu'un intérêt académique, c'est-à-dire sans conséquences. Mais ce n'est pas le cas. Après tout, le défi a été lancé pour respecter les règles linguistiques postulées, et, de toute évidence, la question s'est posée de savoir si le recours au matériel pourrait fournir une solution. La Burroughs Corporation est allée le plus loin dans cette voie avec son ordinateur B5000. On a déjà mentionné l'introduction de descripteurs de données pour accéder aux tableaux et maintenant on ajoutait le descripteur de programme pour les procédures. Les descripteurs de données et programme se distinguaient des mots de données par la valeur de leurs champs indicateurs. Si un descripteur de programme était accédé par une instruction de chargement ordinaire, non seulement un simple accès à la mémoire était exécuté, mais aussi un appel de procédure. En bref, ce qui ne peut pas être décidé par l'analyse du programme, c'est-à-dire à la compilation, doit être fait au moment de l'exécution. Un mécanisme matériel approprié était sollicité en dernier recours. On l'a réalisé par une instruction qui peut être soit une simple accès à des données, soit un appel de procédure complexe, en fonction d'un indicateur dans le mot accédé.

Mais était-ce une bonne ou une mauvaise idée ? C'est plutôt ce dernier cas, non seulement parce que l'ordinateur était compliqué, coûteux et donc moins apprécié sur le marché, mais parce qu'il n'est pas judicieux d'inclure des instructions très complexes, dont le bénéfice est marginal. Après tout, la caractéristique du langage qui a conduit à ces complications doit être considérée comme une erreur, un défaut de conception, et des programmeurs compétents fourniraient de toute façon des spécifications dans l'intérêt de la clarté du programme. L'envie d'obéir à la spécification d'**Algol** à la lettre était une idée d'une sagesse douteuse.

En conclusion de ce sujet, nous présentons une brève déclaration de procédure, simplement pour montrer comment une combinaison de caractéristiques, apparemment innocentes isolément, peut se transformer en puzzle.

```
procedure P(Boolean b; procedure q);
begin integer x;
    procedure Q; x := x+1;
    x := 0;
    if b then P(¬b, Q) else q;
    write(x)
end
```

Quelles valeurs seront écrites en appelant P(true, P) ? Est-ce 0, 1, ou 1, 0 ?

4.8. Les failles de sécurité

L'une des pires caractéristiques jamais rencontrée est la faille de sécurité. L'auteur fait cette déclaration avec un un certain malaise, parce qu'elle a infecté ses langages **Pascal**, **Modula**, et même **Oberon** par ce virus mortel.

La faille de sécurité permet au programmeur de violer la vérification de type par le compilateur. C'est un moyen de dire : « N'intervenez pas, car je suis plus intelligent que les règles ». Les failles prennent de nombreuses formes. Les plus courantes sont les fonctions de transfert de type explicite, telles que

x := LOOPHOLE [i, REAL]	**Mesa**
x := REAL(i)	**Modula**
x := SYSTÈME.VAL(REAL, i)	**Oberon**

Mais on peut aussi les masquer sous la forme de spécifications d'adresse absolue, ou de variantes d'enregistrement comme en **Pascal**. Dans les exemples ci-dessus, on doit interpréter la représentation interne de l'entier i comme un nombre à virgule flottante (réel). Cela ne peut se faire qu'avec la connaissance de la représentation interne des nombres, qu'il ne devrait pas être nécessaire de connaître dans le contexte du niveau d'abstraction fourni par le langage. Dans **Pascal** et **Modula** [4], les failles de sécurité étaient présentes de façon la moins honnête, et dans **Oberon**, elles ne sont présentes que dans un petit nombre de fonctions encapsulées dans un pseudo-système de modules, qui doit être importé et est donc explicitement visible dans l'en-tête de tout module dans lequel ces mécanismes de bas niveau sont utilisés. Cela peut sembler être une excuse, mais cette faille est néanmoins une mauvaise idée.

Pourquoi, alors, ceci a-t-il été introduit ? La raison était le désir de mettre en œuvre des systèmes complets dans un seul et même langage (de programmation système). Par exemple, un gestionnaire de mémoire doit être capable de considérer la mémoire comme un tableau linéaire de données sans type. Il doit être capable d'allouer des blocs et de les recycler, indépendamment de toute contrainte de type. Un pilote de périphérique est un autre exemple pour lequel il est nécessaire de s'affranchir du contrôle de type. Dans les premiers ordinateurs, les périphériques étaient accessibles par des instructions spéciales. Par la suite, les périphériques ont été dotés d'adresses de

mémoire spécifiques. Leurs registres ont été « projetés en mémoire ». C'est ainsi qu'est née l'idée d’autoriser l’attribution d’adresses mémoire absolues à certaines variables (**Modula**). Mais il s'agit d'une facilité qui peut être détournée sous de multiples formes clandestines et grossières.

De toute évidence, l'utilisateur « normal » n'aura jamais besoin de programmer un gestionnaire de mémoire ni un pilote de périphérique et n'a donc pas besoin de ces mécanismes. Cependant - et c'est ce qui fait de la faille une mauvaise idée - le programmeur « normal » a lui aussi le mécanisme à sa disposition. L'expérience a montré qu'il n'hésitera pas à s'en emparer avec enthousiasme comme une merveilleuse caractéristique et l'utiliser partout où c'est possible. C'est particulièrement vrai si les manuels mettent en garde contre son utilisation !

Il reste à dire que la présence d'une faille de sécurité indique généralement une déficience dans le langage proprement dit, elle révèle que certaines choses importantes ne pouvaient pas être exprimées. Un exemple de ce genre est le type ***address*** en **Modula,** qui a dû être utilisé pour programmer des structures de données avec différents types d'éléments. Cela a été rendu impossible par la stratégie de typage strict et statique, qui exigeait que chaque pointeur soit statiquement associé à un type fixe, et ne pouvait que référencer ces objets. Sachant que les pointeurs étaient des adresses, la faille sous la forme d'un transfert de type innocent permettait de laisser les variables pointeurs référencer des objets de tout type. Bien entendu, l'inconvénient était qu'aucun compilateur ne pouvait vérifier l'exactitude de ces affectations. Le système de vérification des types a été annulé, et il aurait aussi bien pu ne pas exister. Souvenez-vous : Une chaîne est aussi forte que son maillon le plus faible.

La solution propre donnée dans **Oberon** [6] est le concept d'extension de type, appelée héritage dans les langages orientés objet. Il est désormais possible de déclarer un pointeur comme référence à un type donné, et celui-ci pourrait pointer tout type qui est une extension du type donné. Cela a permis de construire des structures de données non homogènes et de les utiliser avec la sécurité d'un système de contrôle de type fiable. Une mise en œuvre doit faire la vérification au moment de l'exécution, si et seulement si il n'est pas possible de la faire au moment de la compilation.

Les programmes exprimés dans les langages des années 1960 étaient pleins de failles de sécurité. Ces programmes étaient exposés totalement aux erreurs. Mais il n'y avait pas d'alternative. Le fait qu'un langage comme **Oberon** vous permette de programmer des systèmes entiers à partir de zéro sans exploiter des failles de sécurité (sauf dans le gestionnaire de mémoire et les pilotes de périphériques) marque le progrès le plus significatif en matière de langage sur 40 ans.

5. D'autres techniques

La dernière section des mauvaises idées provient du vaste domaine de la pratique du logiciel, ou plutôt du domaine plus restreint des expériences de l'auteur. Certaines de ces dernières ont été faites il y a 40 ans, mais ce que l'on peut en apprendre est toujours aussi valable aujourd'hui qu'à l'époque. Certaines sont relatives aux pratiques et aux tendances récentes, encouragées pour la plupart par l'abondance de puissance du matériel disponible.

5.1. L'analyse syntaxique

Les années 1960 ont été la décennie de l'analyse syntaxique. La définition d'**Algol** par une syntaxe formelle a fourni les bases nécessaires pour transformer la définition et la compilation des langages en un véritable domaine scientifique. Le concept de compilateur dirigé par la syntaxe s'est affirmé et a donné lieu à de nombreuses activités d'analyse syntaxique automatique sur une base mathématique rigoureuse. On a créé les notions d'analyse ascendante ou descendante, de technique de descente récursive, de mécanismes pour traiter les références en avant et en arrière. Cela s'est

également accompagné d'efforts pour définir la sémantique des langages plus rigoureusement en surchargeant les règles syntaxiques avec les règles sémantiques correspondantes.

Comme c'est le cas dans les nouveaux domaines d'activité, la recherche a dépassé les besoins de la première heure. On a développé des générateurs d'analyseurs de plus en plus puissants qui ont permis de maîtriser des grammaires de plus en plus générales et complexes. Bien que ce soit un processus intellectuel, la conséquence a été moins positive. Elle a conduit les concepteurs de langage à croire que, quelle que soit la construction syntaxique qu'ils aient imaginée, les outils automatiques pourraient sûrement détecter les ambiguïtés, et un analyseur puissant y ferait certainement face. Une idée fausse ! Aucun outil de ce genre ne donnera une indication sur la manière dont cette syntaxe pourrait être améliorée. Non seulement les concepteurs ont ignoré la question de l'efficacité, mais aussi le fait qu'un langage est au service du lecteur humain, et pas seulement de l'analyseur automatique. Si un langage pose des difficultés pour un analyseur, il en va de même pour le lecteur humain. De nombreux langages seraient plus clairs et plus propres, si leurs concepteurs avaient été contraints d'utiliser une méthode d'analyse simple.

Je peux appuyer cette affirmation sur ma propre expérience. Après avoir contribué dans les années 1960 à la mise au point d'analyseurs syntaxiques pour les grammaires de précédence, et les ayant utilisés pour la mise en œuvre des langages **Euler** et **Algol W**, j'ai décidé de passer à une méthode d'analyse plus simple pour le **Pascal**, la méthode de descente récursive. Cette expérience a été des plus encourageantes et je m'y suis tenu jusqu'à ce jour avec une grande satisfaction.

L'inconvénient, si l'on veut appeler ainsi cet avantage, est que l'on doive être beaucoup plus prudent avant la publication et tout effort de mise en œuvre. Cet effort supplémentaire sera plus que compensé par l'utilisation ultérieure à la fois du langage et du compilateur.

5.2. Langages extensibles

Les fantasmes des informaticiens des années 1960 ne connaissaient pas de limites. Encouragés par les succès de l'analyse syntaxique automatique et de la génération d'analyseurs, certains ont proposé l'idée d'un langage souple, ou du moins extensible. L'idée était qu'un programme serait précédé de règles syntaxiques qui guideraient ensuite l'analyseur général tout en analysant les programmes ultérieurs. Un pas de plus : Les règles syntaxiques précéderaient non seulement le programme mais on pourrait les intercaler n'importe où dans le texte.

Par exemple, si quelqu'un souhaitait utiliser une forme personnelle particulièrement fantaisiste de boucle **for**, il pouvait le faire élégamment, en spécifiant même différentes variantes pour le même concept dans différentes sections du même programme. On avait complètement effacé le concept selon lequel les langages servent à communiquer entre les humains, car apparemment chacun pouvait désormais définir son propre langage à la volée. Mais les grands espoirs ont vite été anéantis par les difficultés rencontrées en essayant de spécifier ce que ces constructions privées devraient signifier. Par conséquent, l'idée intrigante des langages extensibles a disparu assez rapidement.

5.3. Procédures imbriquées et table de pointeurs de Dijkstra

Des variables locales pour chaque procédure ! C'est l'une des plus grandes inventions d'**Algol**. L'idée de déclarer d'autres procédures avec des variables locales à ces procédures était naturel et simple. C'était une conséquence naturelle pour tout mathématicien entraîné. Mais parfois, même les concepts les plus simples entraînent des complications importantes pour leur mise en œuvre. L'imbrication des procédures en est un exemple. Le problème réside dans l'adressage des variables (ou des objets plus généralement). Pour le comprendre, il faut quelques préliminaires.

Supposons la configuration suivante de trois procédures imbriquées P, Q et R :

```
procedure P;
begin integer i;
        procedure Q;
        begin integer j;
                procedure R;
                begin integer k; (* P, Q, R, i, j, k visibles ici *)
                end de R;
                (* P, Q, R, i, j visibles ici *)
        end de Q;
        (* P, Q, i visibles ici *)
end de P
```

Comment les symboles i, j et k sont-ils traités dans le corps de R ? La possibilité de récursion empêche l'adressage statique. Chaque appel de procédure entraîne la création d'un nouveau bloc pour ses propres variables locales. On les localise par rapport à l'adresse de base du bloc, de préférence située dans un registre. En général, on place ces blocs dans une pile (la pile des procédures, contrairement à la pile d'expression), et les blocs sont chaînés. Par conséquent, chaque accès parcourt cette chaîne de pointeurs avec un nombre d'étapes qui est donné par la différence de niveau d'imbrication entre la variable accédée et la procédure qui y accède. Dans l'exemple ci-dessus, dans le corps de R, la base de k se trouve en parcourant 0 lien, la base de j en parcourant 1 lien, et la base de i en parcourant 2 liens du chaînage.

Ce schéma simple est, malheureusement, erroné. En plus du *lien dynamique* évoqué, un *lien statique* doit également être géré. Le lien statique pointe toujours vers le bloc de l'environnement statique d'une procédure. Même si cette chaîne est généralement assez courte, l'accès aux variables est ralenti, si une suite de liens doit être parcourue auparavant. E. W. Dijkstra a donc proposé de contourner le parcours d'une liste en mémorisant les pointeurs dans des cellules contiguës en mémoire (ou une table de registres) appelées « *the display* », et d'utiliser le niveau de bloc comme index dans cette table. Le point essentiel de ce système réside dans le fait que cette table de pointeurs doit être mise à jour chaque fois qu'une procédure est appelée et, malheureusement, parfois aussi lorsqu'elle se termine. Ces mises à jour peuvent très bien concerner plusieurs registres, comme le montre l'exemple pathologique suivant :

```
procedure A(procedure X);
begin integer x;
    procedure B;
    begin integer y;
        procedure C; begin integer z; X end de C;
        C
    end de B;
    B
end de A;

procedure P;
begin integer i;
  procedure Q;
  begin integer j;
        procedure R; begin integer k; end de R;
        A(R)
  end de Q;
  Q
end de P
```

Un appel de P provoquera la séquence des activations ultérieures de P, Q, A, B, C, R. Dans R, seuls les blocs de R, Q et P sont accessibles avec les variables respectives k, j, i. À la sortie de R, les blocs C, B, A redeviennent visibles avec les variables respectives z, y, x. Il s'est avéré que la bonne idée avait aggravé le problème plutôt que de le résoudre.

Il est en effet difficile d'estimer, dans des situations aussi complexes, si l'introduction d'une table mémoire rend les programmes plus efficaces grâce à un accès plus rapide à la mémoire (pas de chaînage), ou si elle les ralentit en raison de la surcharge des mises à jour de la table. Ceci dépend fortement de la fréquence de l'accès aux variables par rapport aux appels de procédure. En 1970, nous avions implanté **Pascal** pour l'ordinateur CDC 6000, et il est apparu à l'auteur, qu'en effet, les appels de procédure sont devenus plus lents et le code plus long en raison de la présence d'une telle table.

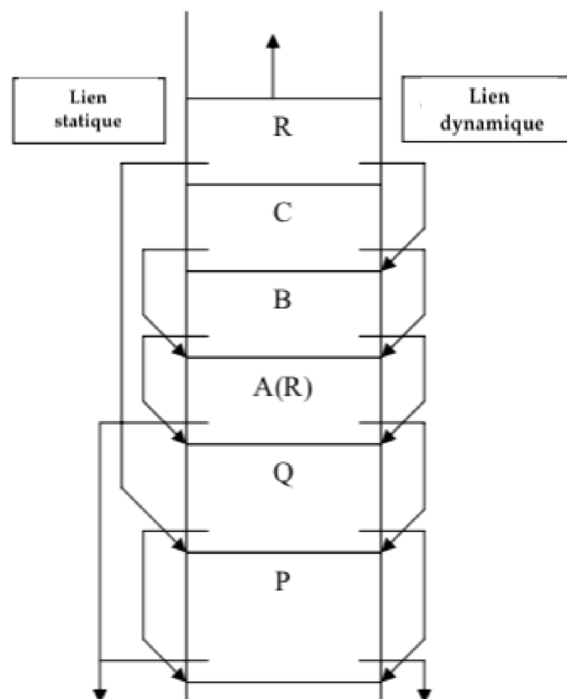


Fig. 2. Pile de blocs d'activation de procédure avec leur chaînage

En 1970, nous avons implanté **Pascal** pour l'ordinateur CDC 6000, On a donc essayé de se passer de la table et on a constaté que le compilateur lui-même était devenu plus rapide et plus court. L'indexation par table était une idée discutable.

Cependant, comme souvent avec les « optimisations », le bénéfice varie selon les différents programmes. Il est évident que la table n'affecte *pas du tout* les programmes sans procédures imbriquées. En effet, nous avons ici une optimisation qui peut rarement être appliquée, dans la mesure où les variables de niveau intermédiaire sont relativement rares. Le compilateur **Algol** Burroughs n'a pas tenu compte de ces variables. Tous les accès devaient être soit globaux, soit strictement locaux. C'était probablement une bonne idée, notamment vis-à-vis de la clarté de programmation.

La principale leçon à retenir est que, lors de la mise en œuvre d'une méthode d'optimisation, il faut d'abord savoir si cela en vaut la peine. En général, elle n'est utile que pour les constructions fréquemment utilisées.

5.4. Les tables des symboles à structure arborescente

Les compilateurs construisent des tables des symboles. On les construit lors du traitement des déclarations, et on les consulte lors du traitement des instructions. Dans les langages qui permettent l'imbrication, on représente chaque bloc d'imbrication par sa propre table.

Traditionnellement, ces tables sont des arbres binaires afin de permettre une recherche rapide. Ayant également suivi sereinement cette longue tradition, l'auteur a osé douter du bénéfice des arbres lors de la mise en œuvre du compilateur **Oberon**. Dès que des doutes surgissent, on est rapidement convaincu que les structures arborescentes ne sont pas utiles pour les entités locales. Dans la majorité des cas, les procédures contiennent une douzaine de variables locales, voire moins. L'utilisation d'une liste linéaire est alors à la fois plus simple et plus efficace.

Dans les programmes d'il y a 30 et 40 ans, la plupart des variables étaient déclarées globalement. Donc, peut-être qu'une structure arborescente était justifiée par la portée globale. Entre-temps, cependant, le scepticisme contre l’intérêt des variables globales avait augmenté. Les programmes d'aujourd'hui ne font pas appel à de nombreuses structures globales, et donc, ici aussi, un tableau à structure arborescente est difficilement recommandable.

Les systèmes de programmation modernes se composent de nombreux modules, dont chacun contient probablement quelques entités *globales* (principalement des procédures), mais pas des centaines. Les nombreux symboles *globaux* des premiers programmes ont été répartis dans de nombreux modules et ne sont pas référencés par un identifiant unique, mais par une combinaison de noms préfixés M.x définissant le chemin de recherche initial. L'utilisation de structures de données sophistiquées pour les tables de symboles était manifestement une mauvaise idée. Nous avions même envisagé des arbres équilibrés !

5.5. L'usage de mauvais outils

Utiliser les mauvais outils est évidemment une idée intrinsèquement mauvaise. Le problème est que, souvent, on ne découvre l'inadéquation d'un outil qu'après avoir investi un effort important pour le construire et le comprendre, l'outil devenant alors « précieux ». C'est ce qui est arrivé à l'auteur et son équipe lors de la mise en œuvre du premier compilateur **Pascal** en 1969.

Les outils disponibles pour écrire des programmes étaient un assembleur, un compilateur **Fortran** et un compilateur **Algol**. Ce dernier a été si mal réalisé que nous n'avons pas osé nous y fier, et l'usage de l'assembleur était considéré comme déshonorant. Il ne restait plus que **Fortran**.

Par conséquent, nos plans naïfs consistaient à construire un compilateur pour un sous-ensemble important de **Pascal** en utilisant le **Fortran**, et une fois terminé, de le traduire en **Pascal**. Ensuite, on utiliserait la technique classique d'amorçage[4] pour compléter, affiner et améliorer le compilateur.

Ce plan s'est cependant effondré face à la réalité. Lorsque la première étape a été achevée après une année-homme de travail, il s'est avéré que la traduction du code **Fortran** en **Pascal** était tout à fait impossible. Ce programme était tellement déterminé par les caractéristiques de **Fortran**, ou plutôt leur absence, qu'il n'y avait pas d'autre option que de réécrire le compilateur. **Fortran** ne comportait pas de pointeurs et d'enregistrements, et les tables des symboles devaient donc être contraintes sous la forme non naturelle de tableaux. **Fortran** ne comportait pas non plus de sous-programmes récursifs. C'est pourquoi la technique complexe d'analyse ascendante par tableau a dû être utilisée avec une syntaxe représentée par des tableaux et des matrices. En bref, les avantages de **Pascal** ne pouvaient être employés que par un nouveau compilateur, complètement réécrit et restructuré.

Cet « incident » a révélé que la voie apparemment la plus facile n'est pas toujours la bonne, mais que les difficultés ont aussi leurs avantages : ce nouveau compilateur, écrit en **Pascal**, ne pouvait être testé pendant le développement, car aucun compilateur **Pascal** n'était encore disponible. Le programme pour compiler au moins un sous-ensemble très important de **Pascal**, a dû être entièrement écrit sans retour d'information de test. C'était un exercice extrêmement sain, et il le serait encore plus aujourd'hui, à l'ère de l'essai rapide et de la correction interactive des erreurs.

Après avoir cru que le compilateur était « complet », un membre de l'équipe a été confiné à son domicile pour traduire le programme dans un langage de bas niveau, dont la syntaxe est vérifiée, et pour lequel un compilateur était disponible. Il est revenu après deux semaines de travail intensif, et quelques jours plus tard, les premiers programmes de test ont été compilés correctement par le compilateur écrit en **Pascal**. L'exercice de programmation consciencieuse s'est avéré

4 Amorçage est utilisé pour traduire bootstrapping.

extrêmement précieux. Jamais programmes ne continrent si peu de bogues, alors qu’aucun outil de débogage n'était disponible !

Par la suite, de nouvelles versions, traitant de plus en plus de constructions de **Pascal**, et produisant du code de plus en plus raffiné, pouvaient être obtenu par amorçage. Immédiatement après le premier programme d'amorçage, nous avons écarté la traduction écrite dans le langage intermédiaire. Son caractère avait été très semblable à celui de l'inquiétant langage **C** de bas niveau, publié un an plus tard. Après cette expérience, il était difficile de comprendre que la communauté du génie logiciel n'ait pas reconnu les avantages de l'adoption d'un langage de haut niveau au typage sûr, au lieu du **C**.

5.6. Les assistants

On a parlé des grands progrès de la technologie de l'analyse syntaxique qui ont commencé dans les années 1960. Les résultats sont toujours présents depuis ces années là. Aujourd'hui, rares sont ceux qui construisent un analyseur syntaxique à la main. Au lieu de cela, on achète un générateur d'analyseur syntaxique et on lui fournit la syntaxe souhaitée.

Cela nous amène au sujet des outils automatiques, que l'on appelle maintenant des assistants. L'idée est qu’ils doivent être considérés comme des boîtes noires, et que l'utilisateur n'aurait pas à comprendre leurs entrailles, car ils ont été conçus et réalisés de manière optimale par des experts. L'idée est qu’ils automatisent des tâches routinières simples, dispensant ainsi les utilisateurs d'ordinateurs de s'en soucier. Les assistants sont censés vous aider - et c'est le point clé - sans que vous ne le demandiez, en tant que fidèles et dévoués serviteurs.

Bien qu'il serait imprudent de lancer une croisade contre de merveilleux assistants, les expériences de l’auteur avec ces assistants furent en grande partie malheureuses. On a pensé impossible d'éviter de les confronter aux éditeurs de texte. Les pires sont ces assistants qui interfèrent constamment avec ce qu'on écrit, qui indentent et numérotent des lignes lorsque cela n'est pas souhaité, qui mettent en majuscule certaines lettres et certains mots à des endroits spécifiques, qui combinent des séquences de caractères en un symbole spécial, ☺, qui convertissent automatiquement une séquence de caractères moins en une ligne continue, etc. Si au moins ils pouvaient être facilement désactivés, mais ils sont généralement obstinés et immortels comme des diables. Tant et tant pour d’intelligents *logiciels pour les nuls* : une mauvaise idée !

6. Les paradigmes de programmation

6.1. La programmation fonctionnelle

Les langages fonctionnels trouvent leur origine dans le langage **Lisp** [2]. Ils ont subi d’importants développements et changements, et ils ont été utilisés pour mettre en œuvre de petits et grands systèmes logiciels. L’auteur a toujours eu une attitude critique à l'égard de ces efforts. Pourquoi ?

Qu'est-ce qu'un langage fonctionnel ou qu'est-ce qui le caractérise ? Il est toujours apparu que c’était leur forme, le fait que l'ensemble du programme consistait à évaluer des fonctions imbriquées, récursives, paramétriques, etc. D'où le terme fonctionnel. Cependant, le cœur de l'idée est que les fonctions n’ont, par nature, pas d’*état*. Cela implique qu'il n'y a ni *variables* ni *affectations*. La place des variables est prise par des paramètres de fonctions immuables, des variables au sens des mathématiques. En conséquence, les valeurs nouvellement calculées ne peuvent pas être réaffectées à la même variable, en écrasant son ancienne valeur. C'est pourquoi la répétition doit être exprimée à l'aide de la récursion. Une structure de données peut au mieux être étendue, mais aucun changement n'est possible dans sa version antérieure. Cela conduit à un degré

extrêmement élevé de recyclage de l'espace mémoire ; un ramasse miettes est l'ingrédient nécessaire. Une mise en œuvre sans ramasse-miettes est impensable.

Poser en postulat un modèle de calcul sans état à partir d'une machine dont la caractéristique première est d'être à état, semble être une idée pour le moins étrange. L'écart entre le modèle et la machine est grand, et donc coûteux à mettre en place. Aucun support matériel ne peut faire oublier ce fait : cela reste une mauvaise idée en pratique. Cela a, en son temps, également été reconnu par les protagonistes des langages fonctionnels. Ils ont introduit les états (et les variables) de diverses manières délicates. Le caractère purement fonctionnel a ainsi a été compromis et sacrifié. L'ancienne terminologie est devenue trompeuse.

Si l'on revient au sujet de la programmation fonctionnelle, il semble que sa véritable pertinence n'était certainement pas son absence d'état, mais plutôt ses structures clairement imbriquées et l'utilisation d'objets strictement locaux. Cette règle peut, bien sûr, être aussi appliquée en utilisant des langages conventionnels et impératifs qui, depuis longtemps, ont souscrit aux notions de structures et fonctions imbriquées et de récursion. Bien sûr, la programmation fonctionnelle implique bien plus que d'éviter les instructions de branchement « **goto** ». On doit également se restreindre à l'usage de variables locales, excepté peut-être quelques variables d'état globales. Elle considère probablement aussi l'imbrication des procédures comme indésirable. L'ordinateur B5000 a apparemment eu raison, après tout, de limiter l'accès aux variables strictement locales et strictement globales.

Les langages fonctionnels constituent-ils donc une catégorie à part entière du seul fait de la terminologie ? Est-ce que leurs fonctions ne sont des fonctions que de forme, mais pas de fond ? Ou est-ce que la substance même du paradigme fonctionnel s'exprime simplement par : « Pas d'effets de bord » ?

Il y a de nombreuses années, et de plus en plus fréquemment, on prétend que les langages fonctionnels sont le meilleur formalisme pour introduire le parallélisme. Il serait plus pertinent de dire : pour faciliter la détection des possibilités de mise en parallèle d'un programme par les compilateurs. Après tout, il est relativement facile de déterminer quelles parties d'une expression peuvent être évaluées en parallèle. Le plus important est que les paramètres d'une fonction appelée puissent être évalués simultanément, à condition, bien sûr, que les effets de bord soient interdits (ce qui ne peut pas se produire dans un vrai langage fonctionnel). Comme cela peut être vrai et peut-être d'un bénéfice marginal, l'auteur pense qu'un moyen plus efficace de permettre à un système de faire bon usage du parallélisme est fourni par l'approche « objet », chaque objet représentant son propre comportement sous la forme d'un processus « privé ».

6.2. La programmation logique

Un autre exemple de paradigme de programmation qui a reçu une large attention est celui de la programmation logique. En fait, il n'existe qu'un seul langage bien connu représentant ce paradigme : **Prolog**. Son idée principale est que la spécification des actions, telles que l'affectation aux variables, est remplacée par la spécification de prédicats sur les états. Si un ou plusieurs paramètres d'un prédicat sont laissés non spécifiés, le système recherche toutes les valeurs d'arguments possibles satisfaisant le prédicat. Cela implique l'existence d'un moteur de recherche de solutions d'énoncés logiques. Ce mécanisme est compliqué, souvent, prend du temps et il est parfois intrinsèquement incapable de fonctionner sans intervention. Cela exige toutefois que l'utilisateur aide le système en fournissant des indications (*cuts*), et il doit donc comprendre ce qui se passe, c'est-à-dire le processus logique d'inférence, précisément la chose qu'on lui avait promis de pouvoir ignorer.

Il faut soupçonner qu'on a vendu un exercice intellectuel intéressant au public en soulevant de grandes attentes. La communauté avait désespérément besoin de moyens pour produire du

meilleur logiciel, plus fiable, et elle était heureuse d'entendre parler d'une possible panacée. Mais les promesses ne se sont jamais concrétisées. Nous rappelons malheureusement les espoirs exagérés qui ont alimenté le projet japonais d'ordinateurs de *Cinquième Génération*, les machines d'inférence **Prolog**. De grandes quantités de ressources y ont été englouties. C'était une idée imprudente et maintenant oubliée.

6.3. La programmation orientée objet

Par opposition à la programmation fonctionnelle et logique, la programmation orientée objet (OOP) repose sur les mêmes principes que la programmation procédurale habituelle. Son caractère est impératif. Un processus est décrit comme une séquence de transformations d'un état. La nouveauté est le découpage d'un état global en *objets* individuels, et l'association des transitions d'état (appelés *méthodes*) avec l'objet lui-même. Les objets sont considérés comme les acteurs qui provoquent la modification de l'état d'autres objets en leur envoyant des messages. On appelle la description d'un modèle d'objet, la définition d'une *classe*.

Ce paradigme reflète étroitement la structure des systèmes « dans le monde réel », et il est donc bien adapté pour modéliser des systèmes complexes au comportement complexe. Sans surprise, la programmation orientée objet trouve son origine dans le domaine de la simulation de systèmes (Simula, *Dahl* et *Nygaard*, 1966). Son succès dans le domaine de la conception de systèmes logiciels parle de lui-même. Son existence a commencé avec le langage **Smalltalk** [5] et s'est poursuivie avec **Object-Pascal**, **C++**, **Eiffel**, **Oberon**, **Java**, **C#**. La première implantation de **Smalltalk** a fourni un exemple de sa pertinence. Cet environnement de programmation a été le premier à comporter des fenêtres, des menus, des boutons et des icônes, des exemples parfaits d'objets (visibles) dans le sens mentionné ci-dessus. Ces exemples ont conduit au succès et à une large adoption. La modélisation directe des acteurs a diminué l'importance de prouver la justesse du programme de manière analytique, car la spécification originale est une question de comportement, plutôt qu'une relation entrée-sortie statique.

Néanmoins, l'observateur attentif peut se demander où se cacherait le cœur du nouveau paradigme, quelle serait la différence essentielle par rapport à la vision traditionnelle de la programmation. Après tout, les anciennes pierres angulaires de la programmation procédurale réapparaissent, bien qu'elles soient ancrées dans une nouvelle terminologie : Les objets sont des enregistrements, les classes sont des types, les méthodes sont des procédures, et l'envoi d'une méthode est équivalent à l'appel d'une procédure. Il est vrai que les enregistrements sont désormais constitués de champs et, en plus, de méthodes. Il est vrai que le concept appelé « héritage » permet la construction de structures de données hétérogènes, qui sont aussi utiles sans orientation objet. Ce changement de terminologie exprimait-il un glissement de paradigme essentiel, ou était-ce un moyen d’attirer l'attention, un « artifice de vente » ?

7. Remarques finales

On a présenté un ensemble d'idées, issues d'un large spectre de la science informatique, et ayant été largement acclamées à leur époque. Pour diverses raisons, une inspection plus détaillée révèle certaines faiblesses. Certaines de ces idées ne sont plus guère adéquates aujourd'hui, en raison des changements et des progrès dans la technologie sous-jacente, d'autres ont été des idées pertinentes qui ont résolu un problème local qui n'est plus actuellement important, et certaines ont reçu une attention et un succès pour diverses raisons non techniques.

Nous pensons qu’on peut apprendre non seulement à partir des mauvaises idées et des erreurs passées, mais plus encore à partir de bonnes idées, en les analysant avec le recul du temps. Ce recueil de sujets peut sembler fortuit, et il est certainement incomplet. De plus, on l'a écrit à partir d'une vision personnelle, avec le sentiment que la science informatique bénéficierait d’une

analyse, d’une critique et surtout d’une autocritique plus fréquentes. Après tout, une autocritique approfondie est la marque de tout sujet se réclamant d'une science.

Références

1. P. Naur (Ed). Report on the Algorithmic Language ALGOL 60. Comm. ACM 3 (May 1960), 299-314.
2. J. McCarthy. Recursive Functions of symbolic Expressions and their Computation by Machine. Comm. ACM 5, (1962)
3. D. E. Knuth. The Remaining Trouble Spots in ALGOL 60. Comm. ACM 10 (Oct. 1967), 611-618.
4. N. Wirth. Programming in Modula-2. Springer-Verlag, 1982.
5. A. Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983.28
6. N. Wirth. The Programming Language Oberon. Software – Practice and Experience, 18, (July 1988), 671-691.

Traduction de l’article original de N. Wirth : « Good Ideas, Through the Looking Glass »
par Jean Conter, Mamoun Filali et Gérard Padiou