15

Deux programmes ennemis s'affrontent à coups de chiffres binaires.

par A. K. Dewdney

Deux programmes informatiques, dans leur habitat naturel que constituent les puces-mémoires d'un ordinateur, se livrent une lutte sans merci, adresse par adresse. Parfois ils partent à la rencontre de l'adversaire ; parfois ils construisent un barrage de bombes numériques ; parfois ils se retranchent eux-mêmes dans une zone moins dangereuse ou s'arrêtent un moment pour soigner leurs blessures. Tel est le jeu que j'ai appelé *Core War* ou La guerre des noyaux. Il se distingue de la plupart des autres jeux sur ordinateurs en ceci qu'il n'y a personne qui joue ! Bien sûr les programmes qui s'affrontent ont été écrits par quelqu'un mais dès lors que la bataille est engagée le créateur du programme ne peut qu'observer passivement sur l'écran la mort ou la survie du produit qu'il a mis des heures à concevoir et à mettre au point. L'issue finale dépend uniquement de celui des deux programmes qui sera le premier touché dans une zone vulnérable.

Le terme *Core War* tire son origine d'une technologie de mémoire aujourd'hui périmée : de 1950 à 1970 environ, la mémoire centrale des ordinateurs comprenait des milliers de petits noyaux magnétiques (*cores*) en forme d'anneaux enfilés sur un tissage de fins fils conducteurs. Chaque noyau avait une capacité de un bit ou unité binaire d'information (0 ou 1). De nos jours les mémoires sont faites avec des puces de semiconducteurs mais dans la langue anglaise le terme *core* évoque le centre vital d'un organisme ou d'un système, et ce terme a été conservé pour désigner la partie de l'unité centrale d'un ordinateur où réside le programme au moment de son exécution.

Les programmes rivaux de *Core War* sont écrits dans un langage spécial que j'ai appelé *Redcode*, et qui fait partie de ce que l'on appelle les langages d'assemblage. Aujourd'hui la plupart des programmes informatiques sont écrits dans un langage de haut niveau et d'utilisation facile comme Pascal, Fortran ou BASIC ; dans ces langages une seule phrase peut commander toute une séquence d'opérations élémentaires de l'ordinateur. Pour qu'un tel programme puisse être exécuté il doit d'abord être traduit en un « code-machine » où chaque instruction est représentée par une longue suite de 0 et de 1. Écrire un programme directement sous forme binaire serait pour le moins très fastidieux.

Les langages d'assemblage se situent à un niveau intermédiaire entre les langages de haut niveau et le code interne de la machine. Dans un programme écrit dans un tel langage chaque phrase correspond habituellement à une seule instruction et se traduit donc par une seule chaîne de chiffres binaires. Plutôt que d'utiliser ces chiffres binaires le programmeur préfère écrire au moyen de courtes abréviations que l'on appelle mnémoniques (parce qu'elles sont plus faciles à retenir que des nombres). La traduction du langage d'assemblage dans le code interne de la machine s'exécute au moyen d'un programme spécial appelé assembleur.

On écrit relativement peu de programmes directement en langage d'assemblage parce que cela conduit à des programmes plus longs et plus difficiles à comprendre et à modifier que les programmes écrits en langage de haut niveau. Il y a cependant certains travaux pour lesquels le langage d'assemblage reste la solution idéale : lorsqu'un programme doit occuper le moins de place possible, ou lorsqu'il doit tourner le plus rapidement possible, on l'écrit généralement en langage d'assemblage. En outre certaines opérations peuvent être exécutées avec un tel langage alors que cela serait impossible avec un langage de haut niveau. C'est le cas, par exemple, quand un programme doit pouvoir modifier ses propres instructions ou se transférer lui-même d'une zone de mémoire dans une autre.

Core War m'a été inspiré par une histoire que j'entendis raconter voici quelques années à propos d'un programmeur malfaisant employé par les laboratoires de recherche d'une grande société que je ne nommerai pas tout de suite. Ce programmeur écrivit un jour, en langage d'assemblage, un programme baptisé *Creeper* qui se dupliquait lui-même chaque fois qu'il était utilisé. Il pouvait aussi se propager d'un ordinateur à un autre à l'intérieur du réseau de la société. Ce programme n'avait pas d'autre fonction que de se reproduire. Au bout de peu de temps *Creeper* avait donné naissance à tellement de répliques de lui-même que des programmes et des fichiers de données beaucoup plus utiles n'arrivaient plus à trouver de place. Cette croissance intempestive ne fut maîtrisée que le jour où quelqu'un eut l'idée de combattre le feu par le feu. Il écrivit un autre programme autoduplicateur que l'on appelle *Reaper* : sa mission était de détruire tous les exemplaires de *Creeper* qu'il pouvait trouver et, à la fin, de se détruire lui-même. *Reaper* fit son travail et tout rentra dans l'ordre.

J'ajoutai foi à cette anecdote, en dépit de quelques lacunes évidentes, sans doute parce que l'affaire m'intéressait. Il me fallut quelque temps pour retrouver la trace des faits authentiques qui avaient donné naissance à cette légende. (J'en dirai un mot à la fin de l'article.) En fait mon désir de croire à cette histoire n'était dû qu'à la fascination exercée par cette image de deux programmes se livrant bataille dans l'obscurité et le silence d'une unité centrale.

L'an dernier je décidai que même si l'histoire s'avérait fausse on pouvait créer quelque chose du même genre. J'imaginai une première version de *Core War*, et avec l'aide de David Jones je la fis fonctionner. Depuis lors nous avons continué à améliorer le programme et nous l'avons amené à un stade tout à fait intéressant.

Core War se compose de quatre parties principales : une mémoire de 8000 mots, le langage d'assemblage *Redcode*, un programme superviseur appelé MARS (initiales de *Memory Array Redcode Simulator*) et l'ensemble des programmes combattants. Deux de ces programmes combattants sont introduits en mémoire à des emplacements choisis au hasard ; aucun d'eux ne sait où se trouve l'autre. Le superviseur MARS pilote le déroulement des programmes suivant la méthode la plus simple, en temps partagé (procédé pour répartir les moyens de calcul entre plusieurs utilisateurs travaillant simultanément). Les deux programmes tournent chacun à leur tour : on exécute une instruction du premier, puis une instruction du second et ainsi de suite.

Ce que fait chaque programme lorsque c'est à son tour de jouer dépend uniquement de ce qu'a écrit le programmeur. Le but est évidemment de détruire le programme adverse en démolissant ses instructions. Cela n'interdit pas une stratégie défensive : un programme peut décider de réparer les dommages qu'il a subis ou de battre en retraite lorsqu'il est soumis à une attaque. Le combat prend fin lorsque le superviseur MARS trouve dans un programme une

instruction qui n'est plus exécutable. Le défaut est présumé être une conséquence de la guerre, et le programme inexécutable est déclaré vaincu.

On peut apprendre beaucoup sur un programme de combat en se contentant d'analyser son contenu, soit mentalement, soit avec un papier et un crayon mais pour mettre le programme à l'épreuve il est indispensable de disposer d'un ordinateur et d'une version de MARS. Il est possible d'adapter ces programmes à des ordinateurs individuels et pour les lecteurs qui souhaiteraient organiser eux-mêmes ces combats, nous avons rédigé une série de directives en un langage proche de Pascal. (Pour obtenir copie de ces directives *en anglais*, envoyer un chèque de 15 francs et une grande enveloppe auto-adressée à *Pour la Science*.)

Avant de décrire *Redcode* et de présenter quelques programmes de combat simples il faut donner certaines précisions sur le réseau de mémoire. J'ai déjà indiqué que la mémoire comportait 8000 adresses mais ce chiffre n'a rien d'impératif ; une mémoire plus réduite peut fort bien convenir. Le réseau de mémoire diffère cependant de ce que l'on trouve habituellement dans les ordinateurs en ceci que son organisation est circulaire : la suite des adresses est numérotée de 0 à 7999 mais elle boucle sur elle-même de sorte que l'adresse 8000 est équivalente à l'adresse 0. Le superviseur MARS interprète toute adresse supérieure à 7999 en ne conservant que le reste de la division par 8000. Ainsi donc, si au cours de la bataille l'un des programmes demande d'attaquer l'adresse 9378, MARS traduit cette adresse en 1378.

Redcode est un langage du type langage d'assemblage, mais simplifié pour cette application particulière. Il contient des instructions pour transférer le contenu d'une adresse vers une autre adresse, pour modifier arithmétiquement le contenu d'une adresse et pour aller chercher l'instruction suivante n'importe où, en avant ou en arrière, à l'intérieur du programme. Un véritable assembleur fournit comme résultat des mots binaires ; ici chaque instruction mnémonique de *Redcode* est traduite par MARS en un grand nombre entier décimal qui est stocké en mémoire ; chacune des adresses du réseau pouvant en contenir un. C'est aussi MARS qui va ensuite décoder ces nombres entiers et déclencher l'exécution des opérations qu'ils représentent.

La figure 1 représente le jeu des instructions *Redcode*. Chaque instruction doit contenir au moins un argument, ou une quantité, mais la plupart des instructions exigent deux arguments. Prenons par exemple l'instruction JMP–7 ; JMP est le code mnémonique pour saut (ou *jump*), il est suivi d'un seul argument : –7. Cette instruction demande à MARS de faire reculer de sept positions l'adresse d'accès à la mémoire, c'est-à-dire d'aller chercher la prochaine instruction 7 positions avant l'instruction JUMP–7 elle-même. Si cette dernière est par exemple l'adresse 3715, la suite du programme doit être cherchée à l'adresse 3708.

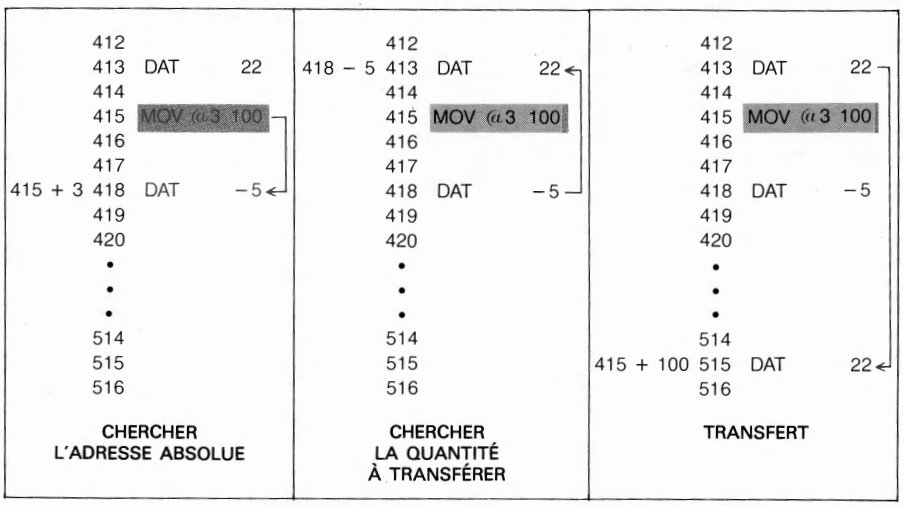
Cette méthode d'accès à la mémoire s'appelle l'adressage relatif ; c'est la seule qui soit employée par *Redcode*. Un programme de combat n'a aucun moyen de connaître sa position absolue dans la mémoire.

L'instruction MOV 3 100 commande à MARS de progresser de trois positions en avant, de lire le contenu et de le recopier 100 positions au-delà de l'instruction MOV elle-même, effaçant du même coup ce qui pouvait se trouver à cette adresse. Dans cette instruction les arguments sont écrits en mode direct, c'est-à-dire qu'ils représentent directement des adresses relatives. Deux autres modes sont également prévus. Si un argument est précédé par le signe @ il devient un argument indirect. Dans l'instruction MOV @ 3 100 l'entier à recopier à l'adresse relative 100 n'est pas le contenu de l'adresse relative 3 mais le contenu d'une adresse elle-même contenue à l'adresse relative 3. (La figure 2 explique plus en détail le mécanisme de l'adressage indirect.) Un argument précédé du signe # indique que cet argument est « immédiat », c'est-à-dire qu'il n'est pas une adresse mais une quantité entière. L'instruction MOV # 3 100 signifie donc qu'il faut écrire le nombre 3 à l'adresse relative 100.

La plupart des autres instructions peuvent se passer d'explications à l'exception de la déclaration de donnée (DAT) qui mérite quelques commentaires. Cette instruction peut servir de mémoire de travail pour conserver une quantité dont le programme aura besoin. En outre ce n'est pas une instruc-

TYPE D'INSTRUCTION	MNÉMONIQUE	CODE	ARGUMENTS		EXPLICATIONS
Move Déplacer	MOV	1	A	B	Transférer contenu adresse *A* à adresse *B*.
Add Ajouter	ADD	2	A	B	Ajouter contenu adresse *A* à adresse *B*.
Subtract Soustraire	SUB	3	A	B	Soustraire contenu adresse *A* de contenu adresse *B*.
Jump Saut inconditionnel	JMP	4	A		Transférer exécution à adresse *A*.
Jump if zero Saut si (B) = 0	JMZ	5	A	B	Transférer exécution à adresse *A* seulement si contenu adresse *B* = 0
Jump if greater Saut si (B) > 0	JMG	6	A	B	Transférer exécution à adresse *A* seulement si contenu adresse *B* > 0
Decrement : jump if zero Décompter (B) : saut si (B) = 0	DJZ	7	A	B	Retrancher 1 du contenu adresse *B* et sauter à adresse *A* seulement si résultat = 0
Compare Comparaison	CMP	8	A	B	Comparer contenus des adresses *A* et *B*, s'ils sont différents sauter l'instruction suivante
Data statement Déclaration donnée	DAT	0		B	Instruction non exécutable, *B* est la valeur de la donnée

1. Jeu d'instructions de Redcode, *le langage d'assemblage pour* Core War.



2. Les trois phases de l'adressage indirect.

tion ; en fait toute cellule de mémoire dont la première position décimale contient un zéro doit être regardée comme contenant une déclaration DAT et par conséquent n'est pas exécutable. Quand on demande à MARS d'exécuter une telle instruction il en est incapable et est donc forcé de déclarer perdant le programme correspondant.

L'entier décimal qui sert à coder chaque instruction *Redcode* peut se découper en plusieurs tranches dont chacune a sa fonction propre (*voir la figure 3*). Le premier chiffre correspond au code mnémonique lui-même, et les deux suivants identifient le mode d'adressage (direct, indirect ou immédiat). Enfin deux tranches de quatre chiffres sont réservées pour les deux arguments. Les arguments négatifs sont représentés par leur complément : –1 sera représenté par 7999 puisque dans la mémoire à organisation circulaire ajouter 7999 à une adresse a le même effet que retrancher 1.

La figure 4 donne la liste des instructions d'un programme de bataille simple appelé *Dwarf*. C'est un programme tout à fait stupide mais très dangereux qui progresse à travers la mémoire en bombardant une adresse sur cinq avec un zéro. Zéro est le nombre entier qui désigne une ligne de donnée, non exécutable comme instruction de programme ; par conséquent, en lâchant un zéro sur un programme ennemi on peut le contraindre à l'arrêt.

Supposons que *Dwarf* occupe les adresses absolues 1 à 4. L'adresse 1 contient initialement DAT–1, mais l'exécution du programme ne peut commencer qu'avec l'instruction suivante : ADD # 5–1. Cette instruction ajoute 5 au contenu de l'adresse précédente c'est-à-dire DAT–1 et la transforme en DAT4. *Dwarf* passe ensuite à l'instruction située à l'adresse absolue 3 : MOV # 0 @ –2. Ce qui veut dire qu'il faut écrire la quantité de 0 à une adresse qui est calculée indirectement comme suit : d'abord MARS décompte deux adresses à partir de 3, ce qui l'amène à 1. Il examine la donnée qui y figure, en l'occurrence le nombre 4 et l'interprète comme une adresse relative par rapport à l'adresse courante. Il compte donc quatre positions à partir de cette adresse courante 1 et inscrit 0 à l'adresse 5.

La dernière instruction de *Dwarf* : JMP–2 crée une boucle sans fin, elle renvoie l'exécution à l'adresse 2, ce qui à nouveau ajoute 5 à la déclaration DAT, la transformant cette fois en DAT 9. Á la prochaine utilisation un zéro sera donc envoyé à l'adresse absolue 10. Les bombes suivantes tomberont aux adresses 15, 20, 25 et ainsi de suite. Le programme lui-même demeure fixe dans la mémoire mais son tir d'artillerie menace tout le réseau. En fin de compte le tir de *Dwarf* progressera jusqu'aux adresses 7990, 7995 et enfin 8000. Pour MARS, 8000 est équivalent à 0 ce qui fait que *Dwarf* échappe de peu au suicide, son projectile suivant tombant à nouveau sur l'adresse 5.

Il est important de réaliser qu'aucun programme de combat, implanté à demeure dans cette mémoire, et occupant plus de quatre lignes ne peut échapper aux attaques de *Dwarf*. L'adversaire n'a que trois possibilités : se déplacer pour tenter d'échapper au bombardement, encaisser les coups et réparer aussitôt les dégâts ou commencer par détruire *Dwarf*. Pour réussir dans cette dernière tentative il faut de la chance : l'adversaire n'a aucune idée de l'endroit où se trouve *Dwarf* dans la mémoire et en moyenne il lui faudra 1600 cycles opératoires avant de réussir à l'atteindre. Si le deuxième combattant est aussi un *Dwarf* chaque programme gagne 30 pour cent des parties et, dans 40 pour cent des combats, aucun des deux programmes ne reçoit de coup fatal.

Avant d'envisager les deux autres stratégies je voudrais présenter un curieux programme de combat d'une seule ligne ; nous l'appelons *Imp*. Le voici :

MOV 0 1

Imp est l'exemple le plus simple d'un programme *Redcode* capable de se déplacer tout seul dans la mémoire. Il copie le contenu de l'adresse relative 0 (c'est-à-dire MOV 0 1) à l'adresse relative 1 qui est la ligne suivante. Lorsque le programme tourne il se propage à travers la mémoire à la vitesse de 1 adresse par cycle, laissant derrière lui un sillage d'instructions MOV 0 1.

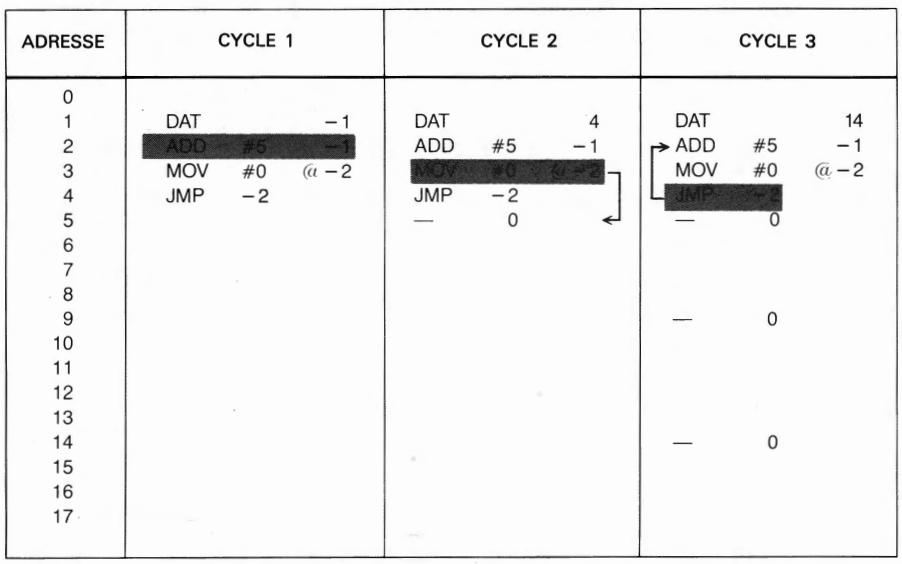
Que se passe-t-il si nous organisons un combat *Imp* contre *Dwarf* ? Le barrage de zéros lâché par *Dwarf* se propage à travers la mémoire plus vite que *Imp* ne se déplace, mais il n'en résulte pas nécessairement que *Dwarf* ait l'avantage. La question cruciale est la suivante : même si le barrage atteint *Imp* est-ce *Dwarf* qui va gagner ? (*voir la figure 5*).

Si *Imp* atteint *Dwarf* le premier, *Imp* va très probablement tailler une brèche dans les instructions de *Dwarf*. Au moment où l'instruction JMP–2 de

MNÉMONIQUE	ARGUMENT A	ARGUMENT B	CODE D'OPÉRATION	MODE D'ADRESSAGE ARGUMENT A	MODE D'ADRESSAGE ARGUMENT B	ARGUMENT A	ARGUMENT B
DAT		–1	0	0	0	0000	7999
ADD	#5	–1	2	0	1	0005	7999
MOV	#0	@ –2	1	0	2	0000	7998
JMP	–2		4	1	0	7998	0000

MODES D'ADRESSAGE		
IMMÉDIAT	#	0
DIRECT		1
INDIRECT	@	2

3. Transformation des instructions de Redcode *en entiers décimaux.*



4. Dwarf, *un programme de combat capable de lâcher un barrage de « bombes zéro ».*

Dwarf provoque un saut de deux pas en arrière, l'instruction qui va se trouver là sera celle de *Imp* MOV 0 1. Résultat : *Dwarf* va être absorbé et va devenir un second *Imp* condamné à poursuivre sans fin le premier tout autour de l'anneau de mémoire. Selon les règles de *Core War* il s'agit là d'une partie nulle. (Notez cependant que ceci n'est que le résultat le plus probable. J'invite les lecteurs à examiner de près les autres éventualités et à découvrir le résultat bizarre de l'une d'entre elles.)

Imp et *Dwarf* appartiennent l'un et l'autre à une catégorie de programmes que l'on peut qualifier de petits et agressifs mais qui ne sont pas intelligents. Dans la catégorie supérieure on peut trouver des programmes plus longs, sans doute un peu moins agressifs, mais assez intelligents pour pouvoir venir à bout des programmes de la première catégorie. Les programmes plus intelligents sont capables d'esquiver une attaque en se transférant eux-mêmes dans une zone à l'abri des risques. Tout programme de ce type inclut une séquence analogue à celle du programme *Gemini* représenté sur la figure 6. *Gemini* n'a pas la prétention d'être un programme de combat complet. Sa seule fonction est de se recopier lui-même à 100 adresses au-delà de son emplacement initial, puis d'enchaîner sur l'exécution de cette nouvelle copie.

Gemini se compose de trois parties principales. On trouve au début deux déclarations de données qui servent de pointeurs : elles indiquent quelle est la prochaine instruction à recopier et à quelle adresse il faut la recopier. Au milieu du programme une boucle fait le travail de copie proprement dit en transportant tour à tour chaque instruction 100 adresses plus loin. À chaque itération de cette boucle les pointeurs des deux premières lignes augmentent d'une unité pour indiquer une nouvelle instruction-source et une nouvelle destination. Dans la même boucle figure une instruction de comparaison (CMP) qui vérifie la valeur du premier pointeur ; quand il a été augmenté neuf fois, la totalité du programme a été recopiée et il faut sortir de la boucle. Il reste encore un détail à régler. L'adresse de copie est celle qui figure dans la deuxième déclaration DAT et sa valeur initiale est 99. Au moment où elle est elle-même transférée elle a déjà été augmentée une fois, de sorte que dans le programme recopié c'est DAT 100 qu'on lit en deuxième ligne. Cette erreur doit être corrigée, ce qui se fait par MOV 99 93. Enfin la dernière instruction ordonne l'exécution de la nouvelle copie.

En modifiant *Gemini* on peut créer toute une catégorie de programmes de combat. L'un d'eux, appelé *Juggernaut*, se recopie lui-même 10 adresses plus loin au lieu de 100. À la manière de *Imp* il tente de tout écraser sur son passage. Toutefois il gagne beaucoup plus souvent que *Imp*, et engendre moins de parties nulles parce qu'un programme situé dans une zone sur laquelle on écrit à nouveau a peu de chances de pouvoir exécuter des fragments du code de *Juggernaut*. Un autre programme nommé *Bigfoot* fonctionne sur le principe de *Gemini*, mais en choisissant un grand nombre premier comme pas de recopie. *Bigfoot* est très difficile à vaincre et a le même effet dévastateur que *Juggernaut* sur les instructions de l'adversaire.

Ni *Bigfoot* ni *Juggernaut* ne sont des programmes très intelligents. Jusqu'ici nous n'avons écrit que deux programmes de combat qui puissent prétendre au second niveau d'intelligence. Ils sont trop longs pour être reproduits ici. L'un d'eux, que nous avons appelé *Raidar*, maintient en permanence deux « sentinelles » qui encadrent le programme lui-même (*voir la figure 7*). Chaque sentinelle se compose de 100 adresses consécutives remplies de 1 et se trouve séparée du programme par un tampon de 100 adresses vides. *Raidar* partage son temps entre des attaques systématiques contre des zones de mémoire éloignées et le contrôle de l'état de ses sentinelles. S'il détecte un changement dans le contenu des zones sentinelles *Raidar* y voit la preuve d'une attaque d'un programme inintelligent tel que *Imp* ou *Dwarf*. Il se recopie alors lui-même de l'autre côté de la sentinelle détruite, reconstitue celle-ci, en établit une nouvelle sur son flanc non protégé, et reprend son activité normale.

Outre la possibilité de se recopier lui-même on peut donner à un programme de combat la possibilité de se réparer lui-même. D. Jones a écrit un programme auto-réparateur qui peut résister à certaines attaques mais pas à toutes. Ce programme s'appelle *Scanner*. Ce programme dispose en permanence de deux copies de lui-même dont une seule est normalement utilisée. La copie active explore périodiquement l'autre pour voir si une attaque extérieure n'a pas modifié certaines instructions. On décèle les altérations par comparaison des deux copies, en supposant toujours que c'est la copie active qui est correcte. Si l'on découvre des instructions incorrectes on les replace

7978	MOV	0	1	
7979	MOV	0	1	
7980	—	0		
7981	MOV	0	1	
7982	MOV	0	1	
7983	MOV	0	1	
7984	MOV	0	1	
7985	—	0		
7986	MOV	0	1	
7987	MOV	0	1	
7988	MOV	0	1	
7989	MOV	0	1	
7990	—	0		
7991	MOV	0	1	
7992	MOV	0	1	
7993	MOV	0	1	
7994	MOV	0	1	} IMP
→7995				
7996				
7997				
7998				
7999				
0				
1	DAT		7994	} DWARF
2	ADD	#5	−1	
3	MOV	#0	@ −2	
4	JMP	−2		
5	—	0		
6				
7				
8				
9				
10	—	0		
11				

5. Bataille Imp *contre* Dwarf. *Qui va gagner ?*

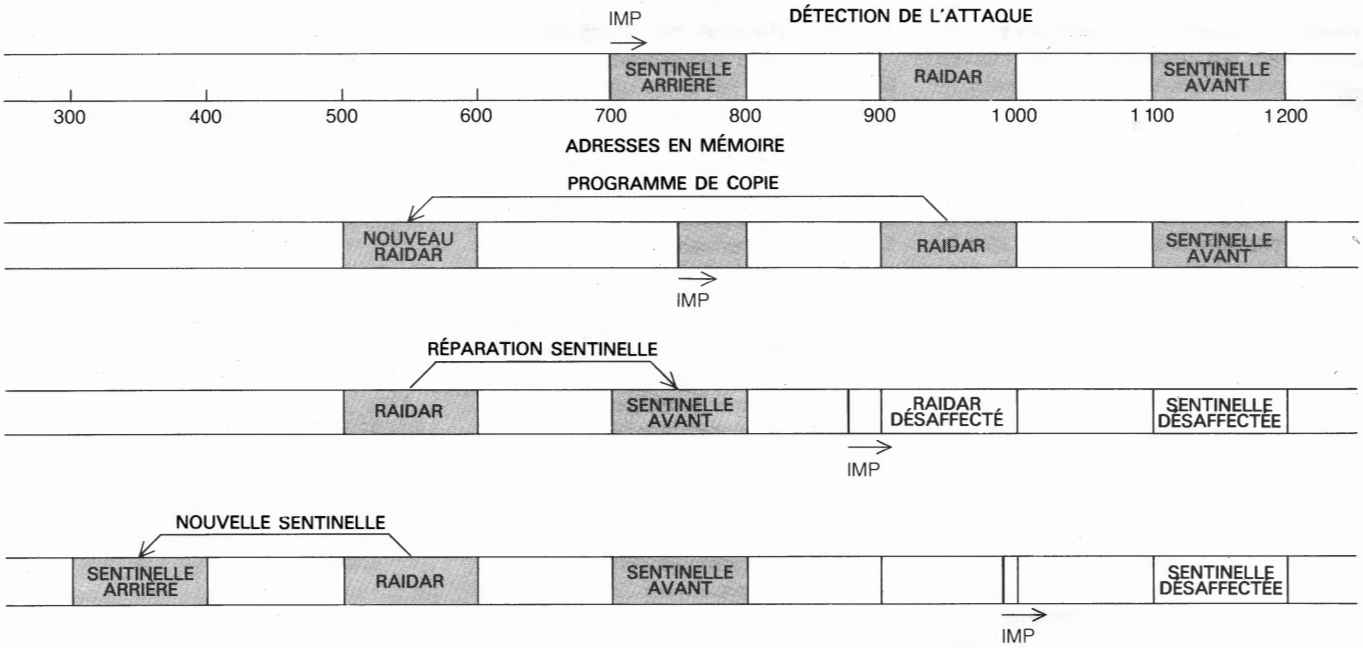
et on passe le relais à la copie de secours qui commence aussitôt à contrôler l'autre.

Jusqu'ici *Scanner* reste un programme purement défensif. Il est capable de résister à des attaques de *Dwarf*, *Imp* et à des agresseurs lents du même type, au moins lorsque l'attaque vient du côté le plus favorable. David Jones travaille en ce moment sur un programme autoréparateur qui conserve trois copies de lui-même.

Je suis curieux de savoir si des spécialistes sauront créer d'autres types de programmes autoréparateurs. On pourrait penser par exemple à conserver deux ou plusieurs copies d'un programme même si l'on n'en utilise jamais qu'une seule. Le programme pourrait comporter une séquence de réparation qui se référerait à une copie de réserve

	DAT		0	– pointeur indiquant l'adresse-origine
	DAT		99	– pointeur indiquant l'adresse d'arrivée
→BOUCLE	MOV	@ −2	@ −1	– instruction de copie
	CMP	−3	#9	– si les dix lignes ont été recopiées...
	JMP	4		– ... alors sortir de la boucle
	ADD	#1	−5	– autrement faire progresser d'un pas l'adresse-origine
	ADD	#1	−5	– ... et également l'adresse d'arrivée
	JMP	−5		– ... puis entrer à nouveau dans la boucle
	MOV	#99	93	– rétablir l'adresse d'arrivée correcte
	JMP	93		– passer à l'exécution de la nouvelle copie

6. Gemini, *un programme capable de se transférer lui-même dans une autre zone de mémoire.*



7. Raidar, *programme de combat perfectionné, capable d'esquiver l'attaque d'un programme plus simple comme* Imp.

pour reconstituer les instructions endommagées. On peut même envisager que cette séquence de réparation sache se réparer elle-même, mais elle resterait sans doute vulnérable à des dégâts subis en certains points précis. Si l'on évalue la vulnérabilité d'un programme en supposant qu'une seule instruction est touchée, combien d'instructions en moyenne sont susceptibles d'entraîner la mort du programme si l'une d'elles est atteinte ? Quel serait de ce point de vue le programme autoréparateur le moins vulnérable qui puisse être écrit ?

Ce n'est que dans la mesure où l'on réussira à développer des programmes relativement robustes que *Core War* pourra devenir un jeu fascinant où l'accent pourra passer des actions défensives aux actions offensives. Les programmes de combat devront alors pourchasser les instructions de l'ennemi, les reconnaître et déployer une attaque intensive là où elles auront été découvertes.

J'ai peut-être donné l'impression que *Redcode* et tout le système MARS sont complètement figés : il n'en est rien. À nos moments de liberté nous avons expérimenté de nouvelles idées et nous restons ouverts à toute suggestion. La vérité c'est que nous avons passé tellement de temps à expérimenter de nouveaux programmes et de nouvelles particularités que certains combats possibles à l'intérieur de notre système actuel n'ont pas encore eu lieu.

Une idée qui fait l'objet de quelques essais est d'ajouter une instruction supplémentaire pour rendre un peu plus faciles l'autoréparation et l'autoprotection. L'instruction PCT *A* protégerait l'adresse *A* contre toute altération jusqu'à la prochaine exécution de l'instruction qu'elle contient. Dans quelle mesure le recours à une instruction de ce genre pourrait-il réduire la vulnérabilité d'un programme ?

Dans les directives que nous avons proposées ci-dessus les lecteurs intéressés trouveront non seulement les règles de *Core War* mais également comment organiser la mémoire et écrire un superviseur MARS dans divers langages de haut niveau. Nous proposons aussi une méthode pour visualiser les résultats du combat. Pour l'instant les règles ci-dessous définissent le jeu avec assez de précision pour permettre aux joueurs qui travaillent seulement sur papier de commencer à concevoir des programmes de combat :

1. Les deux programmes combattants sont chargés en mémoire à des emplacements choisis au hasard mais qui seront initialement séparés par 1000 adresses au moins.

2. MARS déclenche l'exécution d'une instruction de chaque programme en alternance jusqu'à ce qu'il rencontre une instruction inexécutable. Le programme auquel elle appartient a perdu.

3. On peut attaquer un programme avec toutes les armes disponibles. Comme « bombe » on peut se servir d'un zéro ou de tout autre entier, y compris une instruction *Redcode* correcte.

4. Chaque combat se déroule avec une limite de durée qui est fonction de la vitesse de l'ordinateur. Si les deux programmes tournent encore à la fin du temps alloué la partie est déclarée nulle.

L'histoire de *Creeper* et *Reaper* semble bien avoir été inspirée par la combinaison de deux programmes réels. Le premier est un programme de jeu appelé *Darwin* et inventé par Douglas McIlroy des Laboratoires *Bell*. L'autre est un programme nommé *Worm* (ver), qui fut écrit par John Shoch du Centre de Recherches *Xerox* à Palo Alto. L'un comme l'autre datent de plusieurs années, ce qui a laissé tout le temps aux histoires fantaisistes de se propager.

Dans le jeu *Darwin* chaque joueur doit proposer quelques programmes écrits en langage d'assemblage et appelés organismes. Ces programmes sont introduits en mémoire centrale en même temps que ceux des autres joueurs. Les organismes créés par un même joueur (et donc appartenant à la même « espèce ») tentent de tuer ceux des autres espèces et de prendre leur place. Le gagnant est celui dont les organismes sont les plus nombreux lorsque le temps alloué est écoulé. McIlroy a inventé un organisme immortel qui n'aurait pourtant gagné que « quelques parties ». Il était invulnérable mais, semble-t-il, pas très agressif.

Worm était un programme expérimental conçu pour optimiser l'emploi du réseau de miniordinateurs existant chez *Xerox*. Un programme superviseur permettait de charger *Worm* dans des machines inutilisées. Son rôle était de diriger la machine, d'établir la liaison avec les autres *Worm* installés dans les autres machines du réseau et de mettre en oeuvre des programmes d'application importants nécessitant un ensemble multiprocesseur. *Worm* était conçu de façon que quelqu'un qui voulait se servir d'une machine déjà occupée pouvait facilement la récupérer sans perturber les travaux en cours.

On retrouve dans l'histoire de *Creeper* et *Reaper* des éléments aussi bien de *Darwin* que de *Worm*. Avec *Core Ware*, *Reaper* est devenu une réalité.