Habilitation à Diriger des Recherches
Mémoire de Synthèse

Henry Lieberman

Université de Paris VI Pierre et Marie Curie
Laboratoire Informatique Théorique et Programmation
Institut Blaise Pascal
4, place Jussieu, Tour 45-55 203
75252 Paris cedex 05 France

Octobre 1990

Un dimanche après-midi à l'île de la Grande Jatte
Georges Seurat, 1885

Remerciements

Mes remerciements vont à

Jean-François Perrot, d'avoir accepté de présider ce jury, pour son soutien à mon séjour à l'Université Paris 6, et pour son enthousiasme à l'égard de mes recherches,

Patrick Purcell, rapporteur, pour son voyage en France pour participer à ce jury. Je le remercie pour sa confiance, et pour ses idées sur les processus de conception pendant notre travail au Visible Language Workshop,

Harald Wertz, rapporteur, pour son soutien à mon jury, pour l'inspiration de son travail interdisciplinaire sur l'intelligence artificielle, les environnements de programmation et les interfaces utilisateur,

Jean Bézivin, pour sa participation au jury, et pour les discussions sur la programmation par objets,

Pierre Cointe, pour l'invitation de passer une année au sein de son équipe à l'Université Paris 6. Je trouve l'expérience de mon séjour à Paris inoubliable. Je remercie Pierre pour sa relecture soigneuse de ce mémoire. J'apprécie beaucoup son amitié et celle de Isabelle Cointe,

Luc Steels, pour son intérêt à mon travail et sa présence au jury. Je remercie Luc pour des innombrables discussions sur l'intelligence artificielle au MIT, à Paris et à Bruxelles,

Jean-Pierre Briot, Christophe Dony, Philippe Gautron, et Sylvie Lemarié pour leur accueil chaleureux au bureau de RXF/LITP, et leur amitié. Je voudrais les remercier, et aussi Patrick Greussay et Marie-Noëlle Rozin, pour leurs relectures en francais, et leur patience avec les erreurs d'un américain qui rédige son premier article en langue étrangère.

Jacques Sakarovitch, pour son soutien à mon séjour à l'Université Paris 6, et pour son appui dans des situations parfois compliquées,

Gérard Dupoirier et Françoise Bocquillon de Rank Xerox France, pour leur aide pendant mon séjour en France,

Mes étudiants du DEA, car un professeur est toujours un étudiant qui apprend de ses étudiants,

Muriel Cooper et Ron MacNeil, pour leur soutien à mon séjour en France, pour leur accueil au sein du Visible Language Workshop au MIT Media Lab, et pour leurs efforts pour intégrer les mondes de l'art et de l'informatique,

Carl Hewitt, pour son soutien au MIT AI Lab, et pour son rêve de systèmes acteurs,

Hal Abelson et Ira Goldstein pour leur soutien au projet Logo, et leurs idées sur l'éducation,

Nicholas Negroponte, Patrick Winston, Marvin Minsky, Seymour Papert pour la création des environnements de travail au MIT AI Lab et Media Lab qui m'ont permis d'accomplir ces travaux, et pour leur vision de l'avenir de l'informatique,

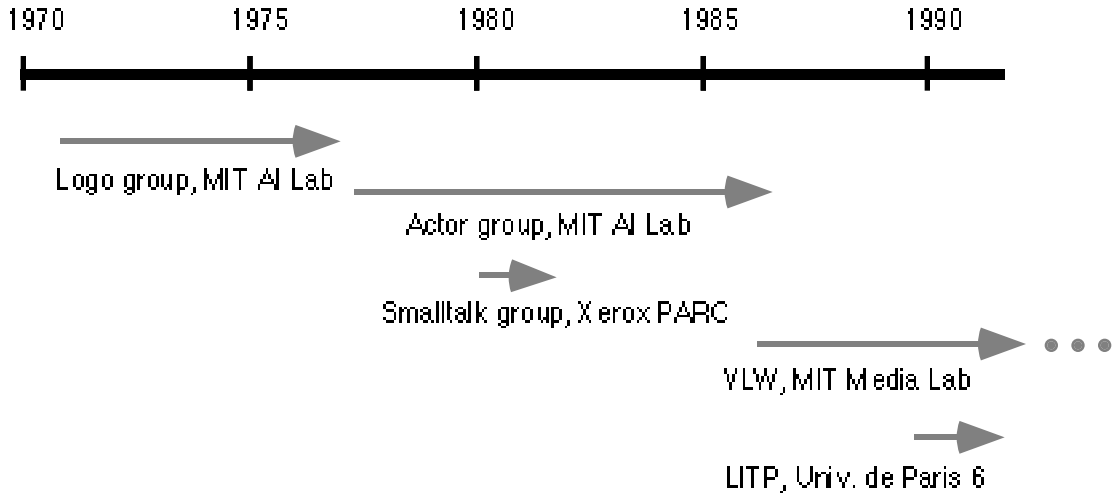
Mes camarades de péniche Josée Giron et Bob Mohl, pour leur foi à la devise de Paris "Fluctuat nec mergitur". Je remercie aussi la Seine.

Ma recherche était parrainée par des subventions de National Science Foundation, DARPA, ONR, Systems Development Foundation, Wang, IBM, Hewlett-Packard, Xerox et NYNEX.

Introduction

Il n'est pas simple de résumer presque vingt années de recherche. Mais dans ce mémoire, je me propose de rappeler l'intégralité de ma carrière comme informaticien. A première vue, les sujets sur lesquels j'ai travaillé peuvent paraître très divers. Néanmoins, il existe une unité fondamentale qui cimente toutes mes recherches. En suivant le chemin qui mène d'un problème de recherche à un autre, il m'arrive de franchir sans y penser les frontières des différentes disciplines comme un voyageur qui n'aperçoit pas une frontière entre deux pays avant de remarquer des panneaux dans une langue étrangère.

A l'exception de mon séjour actuel à Paris, et d'un été passé au Xerox Parc au sein de l'équipe d'Alan Kay, j'ai mené mes recherches de 1971 à 1990 au MIT, à l'Artificial Intelligence Laboratory, et puis au Media Laboratory.



En 1970, je suis venu au MIT comme étudiant, pour étudier la physique. Au MIT, une tradition établie est d'associer très tôt les étudiants aux équipes de recherche. Ainsi je suis devenu membre d'une équipe d'astrophysique au Center for Space Research, où le travail indésirable assigné aux jeunes chercheurs était ... l'informatique. J'ai donc appris à programmer en FORTRAN avec des cartes perforées.

Malgré tout, mon intérêt pour l'informatique s'épanouissait, et j'ai fait la connaissance du célèbre Artificial Intelligence Laboratory. Attiré par la vision révolutionnaire de l'intelligence artificielle et de l'éducation de Seymour Papert, je suis devenu membre de l'équipe Logo, qui travaillait sur les applications de l'intelligence artificielle à l'éducation des enfants. Après avoir obtenu mon diplôme de BS en mathématiques au MIT, je suis resté chercheur dans l'équipe Logo, jusqu'en 1977.

Ensuite, la période la plus longue de ma carrière de chercheur passée au sein d'une même équipe, est celle allant de 1977 à 1987, quand j'ai travaillé sous la direction de Carl Hewitt au MIT Artificial Intelligence Laboratory. J'ai contribué au développement et à la réalisation de la théorie des acteurs, conçu par Hewitt comme une fondation pour les langages de programmation pour l'intelligence artificielle. J'ai travaillé aussi sur de nombreux projets concernant les environnements de programmation et l'intelligence artificielle : le glanage de cellules [*garbage collection*], le dépannage [*debugging*], les interfaces utilisateur et la programmation par l'exemple.

Depuis 1987 je suis membre du Media Laboratory au MIT, un nouveau laboratoire dédié aux technologies de communication intelligentes pour l'avenir. J'appartiens au Visible Language Workshop, une équipe qui réunit des concepteurs graphistes et des chercheurs en intelligence artificielle, travaillant à la conception d'outils graphiques intelligents. Mon déplacement du Artificial Intelligence Laboratory au Media Laboratory m'a permis de passer du statut de chercheur au laboratoire d'intelligence artificielle, spécialiste en graphisme et en

interfaces utilisateur, au statut de chercheur au laboratoire de graphisme et d'interfaces utilisateur spécialiste en intelligence artificielle !

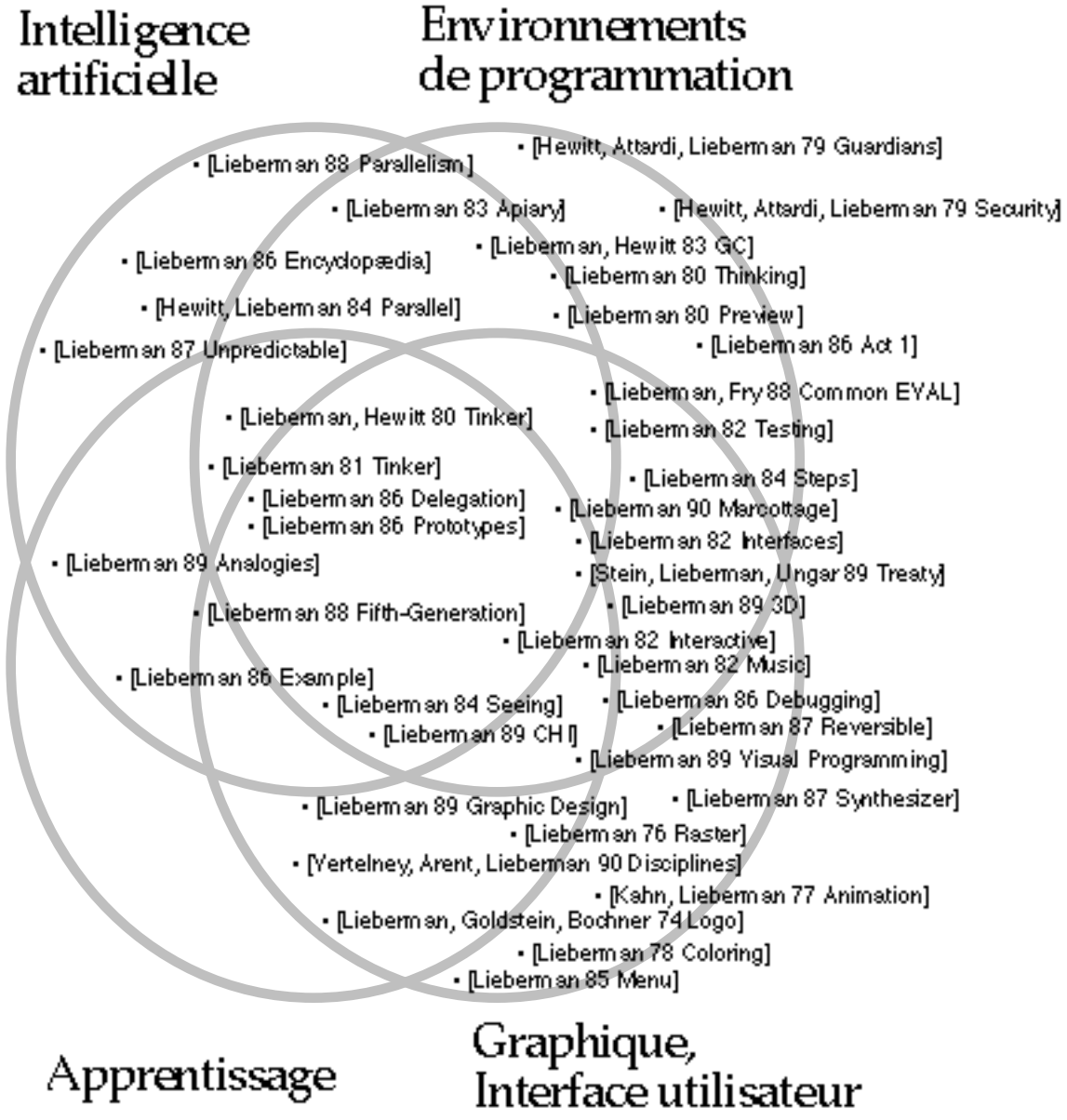
Pendant l'année 1989-1990, j'ai l'honneur d'être accueilli comme professeur invité par l'Université Paris 6. En tant que chercheur, je travaille au sein de l'équipe mixte Laboratoire d'Informatique Théorique et Programmation/Rank-Xerox France de l'Institut Blaise Pascal. En fait, ma collaboration avec le LITP est déjà ancienne, du fait de nos intérêts communs pour la programmation par objets et pour les environnements de programmation.

J'identifie quatre domaines interdépendants de l'informatique que je considère comme pertinents pour mes recherches: l'intelligence artificielle, l'apprentissage, les environnements de programmation et les interfaces utilisateur [y compris le graphisme]. Je ne vois pas ces domaines comme séparés, mais au contraire comme inséparables pour les progrès de l'informatique.

L'intelligence artificielle peut être considérée comme mon objectif au plus haut niveau : transformer les ordinateurs en partenaires intelligents des êtres humains dans leur travail et leur créativité. Mais la réalisation de cet objectif est, malheureusement, encore lointaine. Une grande partie de mon travail est donc consacrée à l'amélioration des environnements de programmation, parce que la programmation est le seul moyen actuel d'augmenter l'intelligence apparente d'un ordinateur.

Mon travail sur les acteurs consistait à réaliser l'environnement de programmation le plus adapté à l'intelligence artificielle. L'apprentissage est intimement lié à l'intelligence artificielle, comme domaine d'application dans mon travail au projet Logo, et comme technique de programmation, dans mon travail sur la programmation par l'exemple. Aujourd'hui, je crois que le goulot d'étranglement des progrès en intelligence artificielle provient du manque de développement des environnements de programmation. Tout comme je crois que la qualité des interfaces utilisateur est le goulot d'étranglement de la programmation. Pourtant, l'amélioration rapide des capacités d'affichage graphique des ordinateurs contemporains est une bonne occasion de faire reculer la "barrière de complexité" qui limite le développement des interfaces utilisateur.

Voici un "plan bibliographique", par lequel j'essaie de situer graphiquement la quarantaine de mes publications scientifiques dans les quatre domaines interconnectés, que sont : l'intelligence artificielle, l'apprentissage, les environnements de programmation et les interfaces utilisateurs. Les catégorisations sont forcément approximatives. Les références intégrales sont données à la fin de ce document.


Intelligence artificielle
Environnements de programmation
Apprentissage
Graphique, Interface utilisateur
• [Hewitt, Attardi, Lieberman 79 Guardians]
• [Lieberman 88 Parallelism]
• [Lieberman 83 Apiary]
• [Hewitt, Attardi, Lieberman 79 Security]
• [Lieberman, Hewitt 83 GC]
• [Lieberman 86 Encyclopædia]
• [Lieberman 80 Thinking]
• [Hewitt, Lieberman 84 Parallel]
• [Lieberman 80 Preview]
• [Lieberman 87 Unpredictable]
• [Lieberman 86 Act 1]
• [Lieberman, Fry 88 Common EVAL]
• [Lieberman, Hewitt 80 Tinker]
• [Lieberman 82 Testing]
• [Lieberman 81 Tinker]
• [Lieberman 84 Steps]
• [Lieberman 86 Delegation]
• [Lieberman 90 Marcottage]
• [Lieberman 86 Prototypes]
• [Lieberman 82 Interfaces]
• [Lieberman 89 Analogies]
• [Stein, Lieberman, Ungar 89 Treaty]
• [Lieberman 89 3D]
• [Lieberman 88 Fifth-Generation]
• [Lieberman 82 Interactive]
• [Lieberman 82 Music]
• [Lieberman 86 Example]
• [Lieberman 86 Debugging]
• [Lieberman 84 Seeing]
• [Lieberman 87 Reversible]
• [Lieberman 89 CHI]
• [Lieberman 89 Visual Programming]
• [Lieberman 89 Graphic Design]
• [Lieberman 87 Synthesizer]
• [Lieberman 76 Raster]
• [Yerteleny, Arent, Lieberman 90 Disciplines]
• [Kahn, Lieberman 77 Animation]
• [Lieberman, Goldstein, Bochner 74 Logo]
• [Lieberman 78 Coloring]
• [Lieberman 85 Menu]

L'époque Logo

Le projet Logo a été initié par Seymour Papert au MIT Artificial Intelligence Laboratory à la fin des années soixante. Il avait pour but d'étudier les applications de l'intelligence artificielle au domaine de l'éducation. Pour ce faire, Papert a conçu le langage de programmation Logo, une version simplifiée de Lisp (le langage préféré de l'intelligence artificielle), destiné aux enfants. A l'époque, l'idée même d'enseigner l'informatique aux jeunes était considérée comme hérétique. Mais notre groupe voulait enseigner aux enfants les nouvelles idées traitant de la résolution de problèmes et de la représentation des connaissances provenant des travaux en intelligence artificielle. Nous pensions améliorer ainsi leur capacité à penser et à raisonner. Ce projet était également très influencé par les théories psychologiques de Jean Piaget sur le développement de l'apprentissage. Ce projet existe toujours, mais se déroule maintenant au sein du Media Laboratory du MIT.

Mon travail sur Logo consistait d'une part à concevoir et à implanter le langage Logo, et d'autre part, à l'enseigner. J'ai travaillé sous la direction de Ira Goldstein, puis de Hal Abelson, à une version de Logo écrite en Lisp [Lieberman, Goldstein, Bochner 74 Logo]. La version de Logo expérimentée à l'école, était écrite en langage machine, pour le mini-ordinateur PDP-11. Elle était donc difficile à modifier et limitée en mémoire. Au contraire, ma version Lisp, réalisée pour le PDP-10, était le véhicule priviligié pour de nouvelles expériences avec le langage, donnant lieu à des projets trop gros pour le PDP-11. Elle permettait par ailleurs d'intégrer Logo aux autres systèmes Lisp du AI Lab.

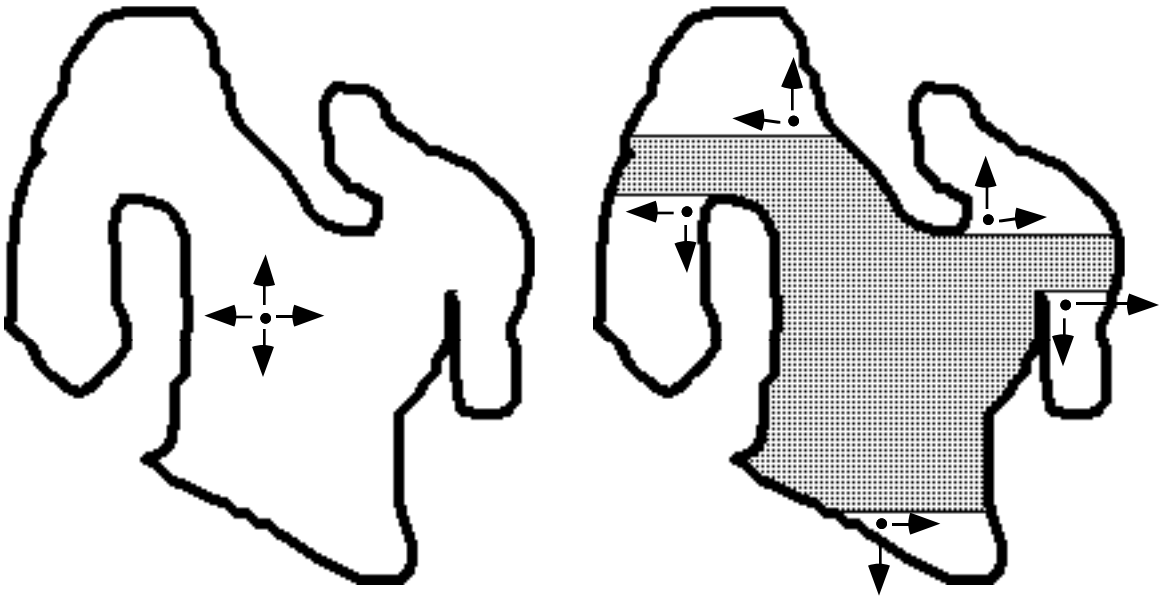
Lisp-Logo est un compilateur de Logo vers Lisp. Il emploie plusieurs techniques dignes d'attention. La syntaxe de Logo, qui supprime les parenthèses de Lisp, ne permet de déterminer l'arité des fonctions qu'au moment de leur exécution. C'est donc un compilateur paresseux. De plus, parce que le code Lisp synthétisé est la seule représentation permanente du programme, c'est un compilateur réversible. La version du source Logo n'est pas conservée, charge étant laissée au scribe PRINT de reconstruire le source dynamiquement. La représentation interne Lisp facilite bien sûr l'intégration avec d'autres systèmes écrits en Lisp. Finalement, ce système Logo intègre aussi un éditeur et un dépanneur.

Beaucoup de projets Logo utilisaient la *tortue*. La tortue est un robot mobile muni d'un stylo pour dessiner sur le sol, controlée par les commandes de déplacement FORWARD [avancez] et RIGHT [tournez à droite]. Les enfants apprennent l'infographie avec un curseur graphique, également appelé la *tortue*, qui accepte les mêmes commandes pour dessiner sur l'écran. Le graphisme jouait donc un rôle essentiel dans le projet Logo. Il éveillait l'intérêt des enfants pour l'informatique, et permettait de leur enseigner les idées importantes de la géométrie et des mathématiques.

Une grande partie de mon travail sur Logo était donc consacré à la réalisation de systèmes graphiques. L'AI Lab a été parmi les premiers laboratoires à obtenir des écrans vidéo à *rasters* qui représentent le contenu de chaque point sur l'écran par un *pixel* de mémoire.

Auparavant, l'infographie était dominée par les *écrans à vecteurs*, où sont tracés de manière répétitive les vecteurs stockés dans une liste de réaffichage. Il est difficile aujourd'hui de s'en souvenir, mais à l'époque la programmation des écrans à rasters requérait une reconsidération radicale d'approche pour les programmeurs habitués aux écrans à vecteurs. La mémoire de l'écran devenait une vraie représentation bidimensionnelle des formes graphiques. L'affichage *2.5-dimensionnelle* résulte d'une simulation des différentes couches des formes graphiques. Elle est réalisée soit par la sauvegarde de la mémoire de l'écran, soit par la fonction XOR. Mon article [Lieberman 76 Raster] qui décrit le premier système graphique Logo pour des écrans à rasters était parmi les premiers à aborder cette transformation conceptuelle de l'infographie. J'ai aussi implanté le premier système graphique Logo en couleurs [Lieberman, Goldstein, Bochner 74 Logo].

J'ai par ailleurs conçu de nouveaux algorithmes adaptés au traitement des images rasters. Mon article [Lieberman 78 Coloring] proposait le premier algorithme de remplissage des formes graphiques présenté dans la littérature. L'algorithme de "l'album à colorier" effectue le balayage interne d'un graphique de forme irrégulière par l'examen localisé des pixels. L'algorithme pourrait aussi servir pour réaliser des robots mobiles aspirateurs! Même si le robot ne dispose pas de la faculté de vision, il est néanmoins assuré de visiter d'une manière efficace chaque recoin d'une pièce.



Un algorithme pour la remplissage d'une forme graphique irrégulière. Le balayage de la forme commence à partir d'une "graine" à gauche. A droite, la graine originelle produit cinq nouvelles graines.

J'ai développé aussi des boîtes à outils adaptées à des programmes scolaires spécialisés, ceux que Papert appelle les *micro-mondes*. Avec Goldstein, j'ai travaillé sur "Germland", un environnement d'automates cellulaires, où l'on peut associer un programme Logo à chaque habitant d'une cellule [Lieberman, Goldstein, Bochner 74 Logo]. Une autre boîte à outils de Lisp-Logo est consacrée à la synthèse de la musique par ordinateur. J'ai travaillé brièvement avec David Marr sur un système qui produit des images "stéréogrammes pointillées aléatoires" [*random-dot stereograms*], un outil pour des expériences en vision tridimensionnelle. J'ai enseigné la théorie des couleurs aux lycéens avec l'aide d'un système pour la simulation de mixage additif et soustractif des couleurs. Un système d'animation facilite la création de graphique dynamique à partir de scénarios [Kahn, Lieberman 77 Animation].

Mes recherches sur Logo m'ont permis d'acquérir une expérience très profitable pour mon travail postérieur. Je suis resté très intéressé par tout ce qui concerne l'éducation et l'intelligence artificielle. La relation entre ces deux domaines m'a conduit plus tard à étudier le problème de l'apprentissage par ordinateur, d'où mon travail sur la programmation par l'exemple. Mon expérience de la réalisation de Lisp-Logo m'a permis d'acquérir la pratique de l'implantation de langages de haut niveau en Lisp. Plus tard, j'ai profité de cette expérience pour mon travail d'implantation des systèmes d'acteurs. Depuis cette expérience, je me suis rendu compte de l'importance de l'environnement de programmation dans son intégralité, en dépassant le simple cadre du langage de programmation. Le travail sur les systèmes graphiques m'a convaincu de l'importance de la communication visuelle entre l'ordinateur et l'utilisateur. L'enseignement de l'informatique aux enfants m'a sensibilisé à la nécessité de la simplicité d'apprentissage et d'utilisation des interfaces.

Des tortues aux acteurs

Pour enseigner aux élèves les idées fondamentales de Logo, nous avons employé une approche qui consistait à *anthropomorphiser* les éléments du système. Ainsi, nous avons encouragé les élèves à imaginer la tortue, les fonctions et les variables Logo comme de petites personnes. Chaque personne peut répondre à certaines demandes [les opérations du langage de programmation], et émettre des requêtes aux autres personnes. A toute personne est assignée sa propre tâche, et le programme fonctionne comme une société. Nous avons même encouragé les élèves à jouer le rôle de la tortue ou d'autres éléments du programme pour mieux comprendre leur fonctionnement.

Carl Hewitt est l'un des premiers qui ait eu l'intuition d'appliquer cette métaphore anthropomorphique pour réaliser les fondations de nouveaux

langages pour l'intelligence artificielle. L'approche *acteur* était née! Trouvant cette idée passionnante, j'ai rejoint son équipe dès 1977. J'ai pu alors mener à bien de nombreuses réalisations et développements dans le domaine des systèmes d'acteurs.

Un *acteur* est un objet actif qui communique avec les autres acteurs par envoi de messages. Chaque message est également, lui-même, un acteur. Parce qu'il n'existe qu'une seuel sorte d'objet (l'acteur) et qu'il n'existe qu'une seule structure de contrôle (le message), les acteurs réalisent un modèle de calcul simple et élégant pour étudier les fondements de l'informatique. Néanmoins, le modèle acteur est suffisamment général et abstrait pour modéliser n'importe quel aspect d'un langage classique. L'absence d'état global ou de séquencement implicite permet aux acteurs de modéliser très naturellement le parallélisme. Les acteurs conduisent à une extrême modularité, car chacun d'entre eux possède sa propre connaissance et sa propre expertise. Grâce à son caractère dynamique et sa souplesse, le modèle acteur est bien adapté aux besoins de la programmation en intelligence artificielle.

Du point de vue pédagogique, j'ai contribué par des articles d'introduction aux techniques de programmation par objets, à promouvoir leur utilisation dans l'intelligence artificielle. J'ai ainsi publié des textes pour l'Encyclopédie d'intelligence artificielle de Stuart Shapiro [Lieberman 86 Encyclopædia], pour une revue scientifique destinée au grand public [Kahn, Lieberman 77 Animation] et pour le Computer Music Journal [Lieberman 82 Music].

L'interprète acteur Act 1

Au sein de l'équipe acteur, j'ai d'abord été chargé de la réalisation des interprètes acteurs écrits en Lisp. J'ai commencé par étudier l'interprète PLASMA de Richard Steiger, Howie Shrobe et al, puis l'interprète CCH de Todd Matson. Peu après, j'ai décidé de recommencer une nouvelle implantation, que j'ai nommé *Act 1*. Act 1 était écrit en MacLisp pour le PDP-10, puis pour la Lisp Machine. J'ai décrit ce langage et son interprète pour la première fois dans [Lieberman 81 Preview]. Les aspects de cette implantation qui touchent au parallélisme sont décrits dans [Lieberman 81 Thinking]. La description d'Act 1 la plus complète et la plus accessible est publiée dans le livre édité par Yonezawa et Tokoro consacré aux objets concurrents [Lieberman 86 Act 1].

Le but de cet interprète était de créer un environnement de programmation dans lequel *tout*, y compris les fonctions et les données d'un système classique, serait représenté uniformément par des acteurs. Comme corollaire, tout calcul serait produit par l'envoi des messages entre acteurs. A l'époque, il n'était pas du tout évident qu'un tel but soit réalisable. On s'est souvent moqué de notre approche et combien de fois n'a-t-on pas dit : "Hewitt et Lieberman veulent avoir un acteur pour chaque bit dans la machine !".

Pour éviter le problème de récursion et de régression infinie, j'ai introduit la notion d'"acteur atomique" [*rock-bottom actor*]. La représentation d'un acteur telle qu'elle est définie par le programmeur se compose d'une partie procédurale, le scénario [*script*], et d'une partie déclarative, les accointances [*acquaintances*]. Mais la représentation d'un acteur atomique, comme un nombre ou une chaîne de caractères est directement un objet Lisp. Normalement, quand l'interprète acteur envoie un message, il fait appel au scénario, en rendant accessible au scénario les accointances et le message. Mais dans le cas d'un acteur atomique, l'interprète est modifié pour chercher le scénario associé au type Lisp de l'objet. L'objet Lisp est lui-même considéré comme accointance unique. Le code associé aux acteurs atomiques se trouve accordé la permission de "tricher" en faisant appel directement aux fonctions primitives de Lisp, à condition que cette violation du protocole de l'envoi de messages ne soit pas visible du programmeur. Cette innovation permet d'étendre le modèle d'envoi de messages au plus bas niveau du système, et facilite ainsi l'intégration avec le langage hôte Lisp.

Les interprètes acteurs antérieurs à Act 1 n'avaient pas réussi à intégrer correctement les types primitifs Lisp. Soit ils ont représenté explicitement les scénarios des types primitifs avec de lourdes conséquences pour l'efficacité, soit ils n'ont pas accordé le comportement de réception de messages aux types primitifs Lisp. Smalltalk et Simula étaient implantés directement en langage machine, sans un langage hôte de haut niveau comme Lisp.

Une des fonctions essentielles d'un interprète acteur est la décomposition des structures de contrôle du langage de programmation en tâches primitives : la création d'acteurs et l'envoi de messages. La traduction d'un appel de fonction en messages unidirectionnels nécessite l'introduction des acteurs *continuations*. La continuation REPLY-TO reçoit le résultat d'un appel de fonction et reprend le calcul. La continuation COMPLAIN-TO permet quant à elle, une gestion des erreurs. La séparation des continuations REPLY-TO et COMPLAIN-TO rend la gestion d'erreurs indépendante de la "pile" de continuations associée aux appels de fonctions et aux structures de contrôle parallèles.

Une bonne partie d'un interprète acteur est traditionnellement consacrée à la tâche de traduction de continuations. C'était en particulier le cas pour PLASMA. Pour Act 1, je me suis servi des macros Lisp comme d'une sorte de compilateur incrémental de continuations. Le mécanisme de traduction des continuations, caché par les macros, m'a permis de produire un code *méta-circulaire*, beaucoup plus concis et plus clair à lire. La méta-circularité de l'implantation donne accès au niveau inférieur de *transmission* pour lequel les continuations et l'environnement sont visibles, et à la fois au niveau supérieur correspondant aux appels de fonctions.

L'interprète Act 1 est réparti parmi les acteurs qui représentent les éléments du langage Act 1. Le message EVAL correspond à l'évaluateur classique. Le langage peut être étendu par l'introduction d'un nouvel objet ayant son propre comportement en réponse à EVAL. Mais il existe aussi un message MATCH, dédié au filtrage [*pattern-matching*] employé en Act 1 pour la réception des arguments. Les macros Act 1 servent aussi à réaliser ce filtrage d'une manière

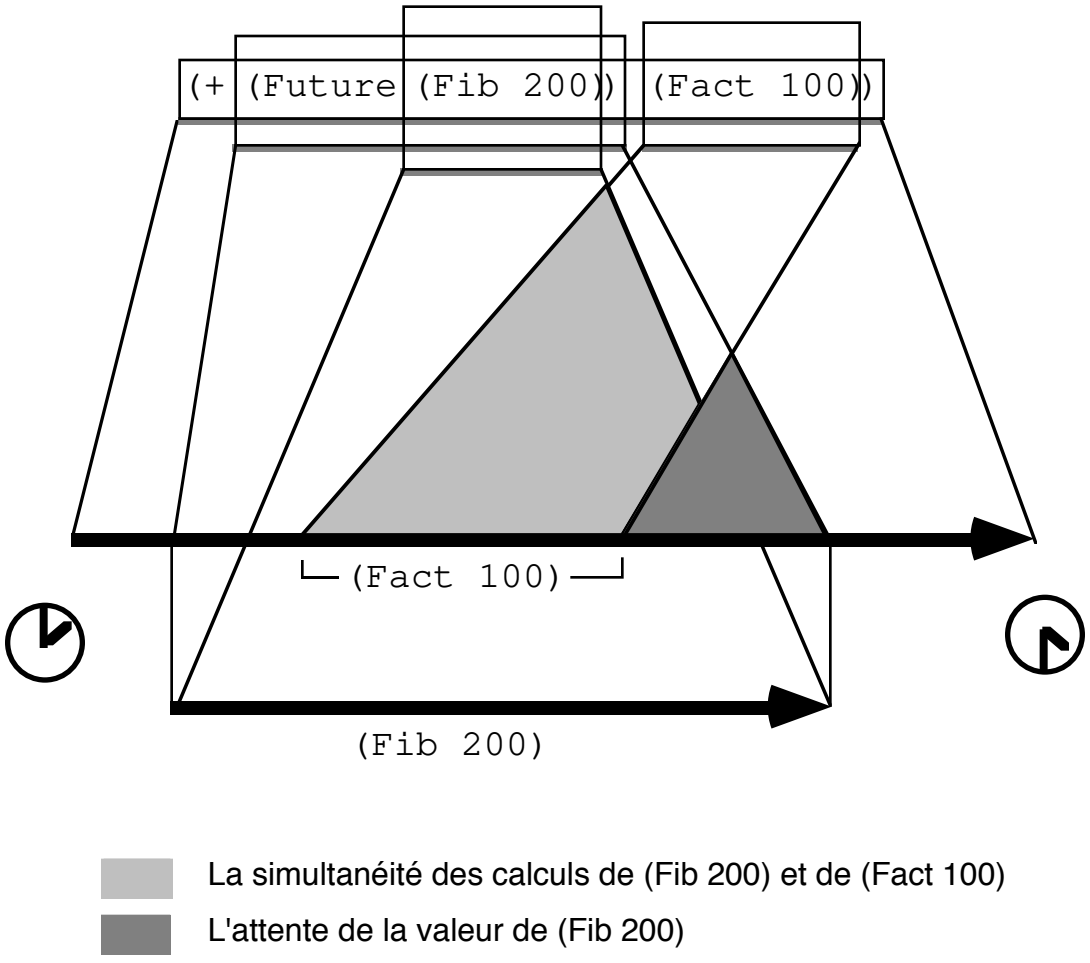
méta-circulaire. Beaucoup d'autres fonctionnalités d'un interprète classique sont réalisées par les scénarios des objets Act 1. Les opérations d'entrée-sortie READ et PRINT sont réalisées d'une façon telle qu'elles soient réciproques, l'une de l'autre. Le message de comparaison EQUAL et le message de vérification des types ARE-YOU sont réalisés par un accord entre l'émetteur et le récepteur. Act 1 a beaucoup contribué aux techniques d'interprétation des langages à objets, d'autant plus que les autres langages à objets de l'époque (Smalltalk et Simula), n'existaient qu'en versions compilés.

Le parallélisme en Act 1

Peut-être le plus grand intérêt de l'interprète Act 1 réside dans son traitement du parallélisme. A la différence d'autres langages à objets comme Smalltalk et Simula, qui réalisent un pseudo-parallélisme par le transfert explicite de contrôle, Act 1 profite du parallélisme implicite du modèle acteur. A la différence d'autres modèles de parallélisme tel que CSP de Hoare, Act 1 permet la gestion dynamique des processus.

L'introduction et le contrôle du parallélisme sont respectivement la responsibilité des acteurs futurs [*futures*] et des acteurs synchronisateurs [*serializers*]. Comme tout autre acteur, les futurs et les synchronisateurs peuvent être créés dynamiquement. Comme eux, ils sont récupérés à la fin de leur vie par le glaneur de cellules. La notion de futur est due à Henry Baker, et elle trouve sa première implantation dans mon interprète Act 1. L'idée de synchronisateur est due à Carl Hewitt, et ses propriétés sont discutées dans nos articles [Hewitt, Attardi, Lieberman 79 Guardians] et [Hewitt, Attardi, Lieberman 79 Security].

Un acteur futur représente le résultat d'un calcul parallèle. Plus précisément, un futur réalise la promesse de livrer le résultat d'un calcul en cours. Normalement, quand on évalue une expression, la continuation de retour attend la fin de l'évaluation pour reprendre le calcul. Mais quand on évalue une expression futur la continuation reçoit immédiatement un acteur futur qui représente cette valeur, et le calcul de l'expression peut continuer en même temps. La valeur définitive de l'acteur futur n'est pas indispensable avant l'envoi d'un message vers le futur. A ce moment, le message est obligé d'attendre la fin du calcul. Quand le calcul est terminé, le message est envoyé à la valeur définitive. Parce que le comportement du futur est donc indifférencable de sa valeur, le futur réalise ainsi le parallélisme de manière transparente.



Un diagramme du calcul parallèle d'un futur. Chaque sous-expression est associée avec un intervalle de temps sur les axes. Le calcul de (FIB 200) et celui de (FACT 100) se déroulent simultanémen. Le calcul de la somme est obligé d'attendre la fin du calcul de FIB.

Si le futur est l'acteur qui engendre les processus parallèles, le synchronisateur est celui qui restreint le parallélisme. Un synchronisateur est un acteur qui établit des contraintes sur l'ordre de réception des messages. Le but d'un synchronisateur est de protéger un état interne contre des modifications incohérentes venant simultanément de processus multiples indépendants.

Act 1 dispose de plusieurs variétés de synchronisateurs. Le plus simple est ONE-AT-A-TIME, qui est l'analogue du moniteur d'un langage parallèle classique. ONE-AT-A-TIME est un acteur qui ne permet qu'un seul message à la fois. Lorsqu'un message est reçu, les autres messages venant d'autres processus parallèles sont obligés d'attendre la fin du calcul pour être traités. Un tel synchronisateur suppose l'existence d'une file d'attente de messages. Mais ONE-AT-A-TIME est limité aux applications pour lesquelles l'ordre de réponse est strictement déterminé par l'ordre de réception de messages.

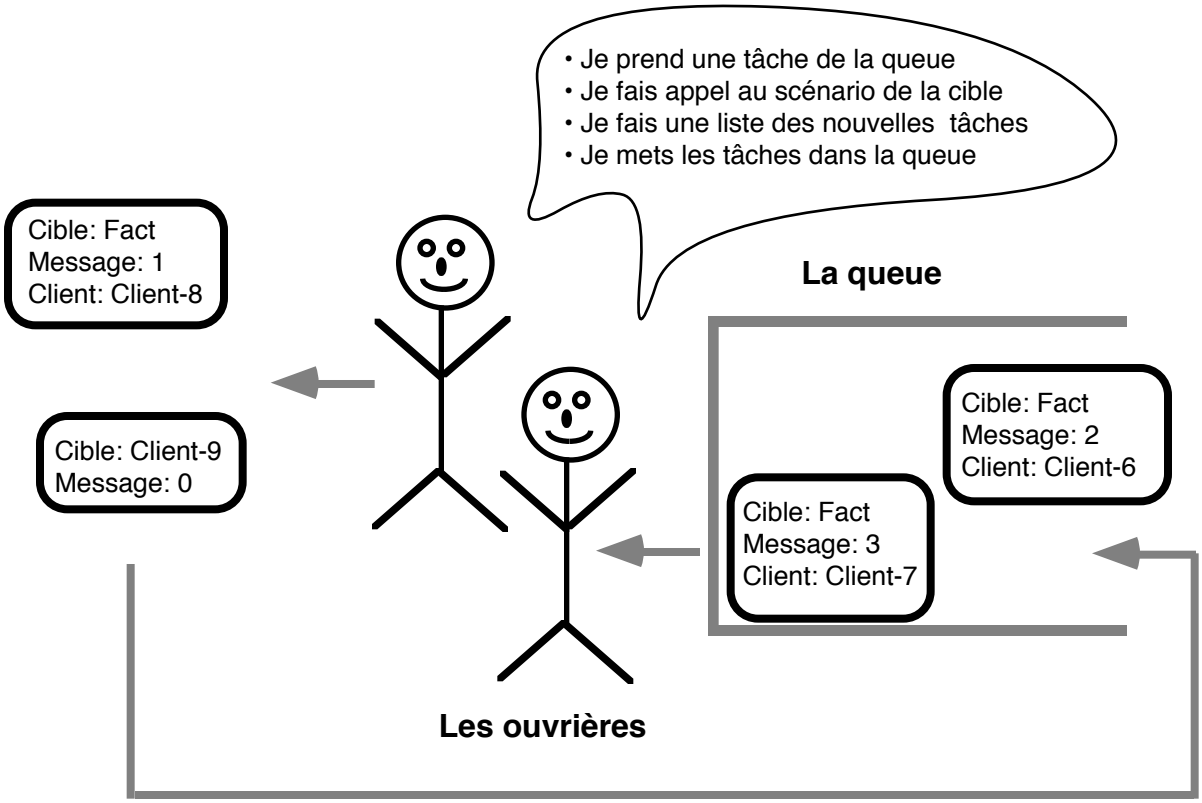
Un synchronisateur plus général est GUARDIAN. GUARDIAN permet de retarder la réponse à un message, et de continuer à recevoir d'autres messages. Mais GUARDIAN exige une gestion explicite des continuations. ONE-AT-A-TIME suffit pour une application comme l'exclusion mutuelle dans une base de données, tandis que GUARDIAN est nécessaire pour des applications comme la programmation d'une agence de placement, pour lesquelles une demande peut être satisfaite par un message reçu ultérieurement. La technique d'implantation d'acteurs futurs et synchronisateurs en Act 1 emploie les salles d'attente [*waiting rooms*]. Ce sont des listes qui mémorisent les continuations afin de suspendre le calcul, par exemple pour un message envoyé à un futur qui est encore en train de calculer.

L'acteur RACE est une généralisation parallèle des listes en Lisp. Si CONS construit des listes qui respectent l'ordre dans lequel les arguments lui sont passés, RACE évalue ses arguments en parallèle et engendre une liste dont l'ordre des éléments est fonction de la terminaison du calcul de leurs valeurs. RACE est utile pour les disjonctions parallèles, ou la concaténation des séquences de générateurs [*stream merge*].

L'évolution des acteurs : d'Act 1 à l'Apiary

La deuxième génération de réalisations d'acteurs sur laquelle j'ai travaillé devait rendre plus proche l'objectif de la réalisation une vraie machine acteur. L'*Apiary* [rucher] est un système acteur écrit en Lisp pour un réseau de machines Lisp Symbolics, reliées par un réseau Ethernet. Si Act 1 simule le parallélisme pour un seul processeur, l'Apiary exécute ses programmes physiquement en parallèle, en les répartissant sur le réseau. Ce système est décrit dans [Lieberman 83 Apiary]. Plus généralement, nous avons réfléchi aux conséquences de l'introduction du parallélisme pour l'avenir des systèmes pour l'intelligence artificielle dans [Hewitt, Lieberman 84 Parallel].

A la différence d'Act 1, qui réalise les acteurs primitifs directement dans le langage hôte Lisp, l'Apiary est une simulation d'une machine acteur. Les processeurs acteurs, appelé ouvrières [*workers*] par analogie aux abeilles, sont simulés, et peuvent fonctionner sur une ou plusieurs machines Lisp. Le système partage dynamiquement les ressources entre processeurs, grâce à ses algorithmes de diffusion : la *migration* répartit le stockage des acteurs parmi les processeurs et l'équilibrage de travail [*load-balancing*] diffuse les tâches primitives. Les acteurs primitifs sont compilés vers les objets qui représentent les "instructions" d'un processeur acteur. Ces instructions correspondent aux quatre capacités fondamentales du modèle acteur. Un acteur peut créer de nouveaux acteurs, envoyer des messages, faire appel à des fonctions primitives (les conditionnelles, l'addition des nombres...), ou remplacer son comportement par un nouvel acteur (dans le cas d'un synchronisateur).



L'Apiary comprend un langage de haut niveau pour l'écriture de scénarios, Scripter. Comme les macros d'Act 1, Scripter est chargé de la traduction des appels de fonctions en envois de messages et continuations. L'Apiary intègre aussi un dépanneur parallèle et multi-fenêtré qui affiche l'état instantané de chaque processeur Apiary. Le développement de l'Apiary continue aujourd'hui au MIT AI Lab, avec le système Acore et son dépanneur graphique Traveler de Carl Manning.

Plus récemment, mon intérêt pour le parallélisme s'est orienté vers ses applications aux domaines des interfaces et de l'intelligence artificielle. L'arrivé ces prochaines années des ordinateurs parallèles bouleversera sans doute profondément la résolution de problèmes par l'ordinateur et la communication entre la machine et l'utilisateur.

[Lieberman 87 Unpredictable] discute le fait que le parallélisme et les méthodes heuristiques d'intelligence artificielle introduisent un facteur indispensable : *l'imprévisibilité* du calcul. D'apparence contradictoire, le principe de la *préparation pour l'imprévisible* est un guide pour la conception de nouveaux systèmes parallèles en intelligence artificielle.

L'avènement du parallélisme obligera, pour les systèmes de représentation de connaissances, à une reconsidération de leurs stratégies de recherche, d'inférence, et d'interaction avec l'utilisateur. [Lieberman 88 Parallelism] procède à une

analyse de ce problème, et suggère les premières solutions. J'observe en particulier, qu'une présomption de séquentialité dans les interprètes associés à ces systèmes, les empêche souvent de profiter vraiment du parallélisme intrinsèque. L'introduction du parallélisme dans l'interface est également nécessaire pour que ces systèmes profitent au maximum du parallelisme potentiel.

La délégation et le problème du partage de la connaissance

Le problème du partage de la connaissance et de l'expertise est une question fondamentale pour les langages à objets et pour l'intelligence artificielle. La technique classique d'héritage est employée par les systèmes à classes, comme Smalltalk et Simula. Mon travail sur les acteurs a mis en évidence le mécanisme alternatif de *délégation*, peu apprécié auparavant.

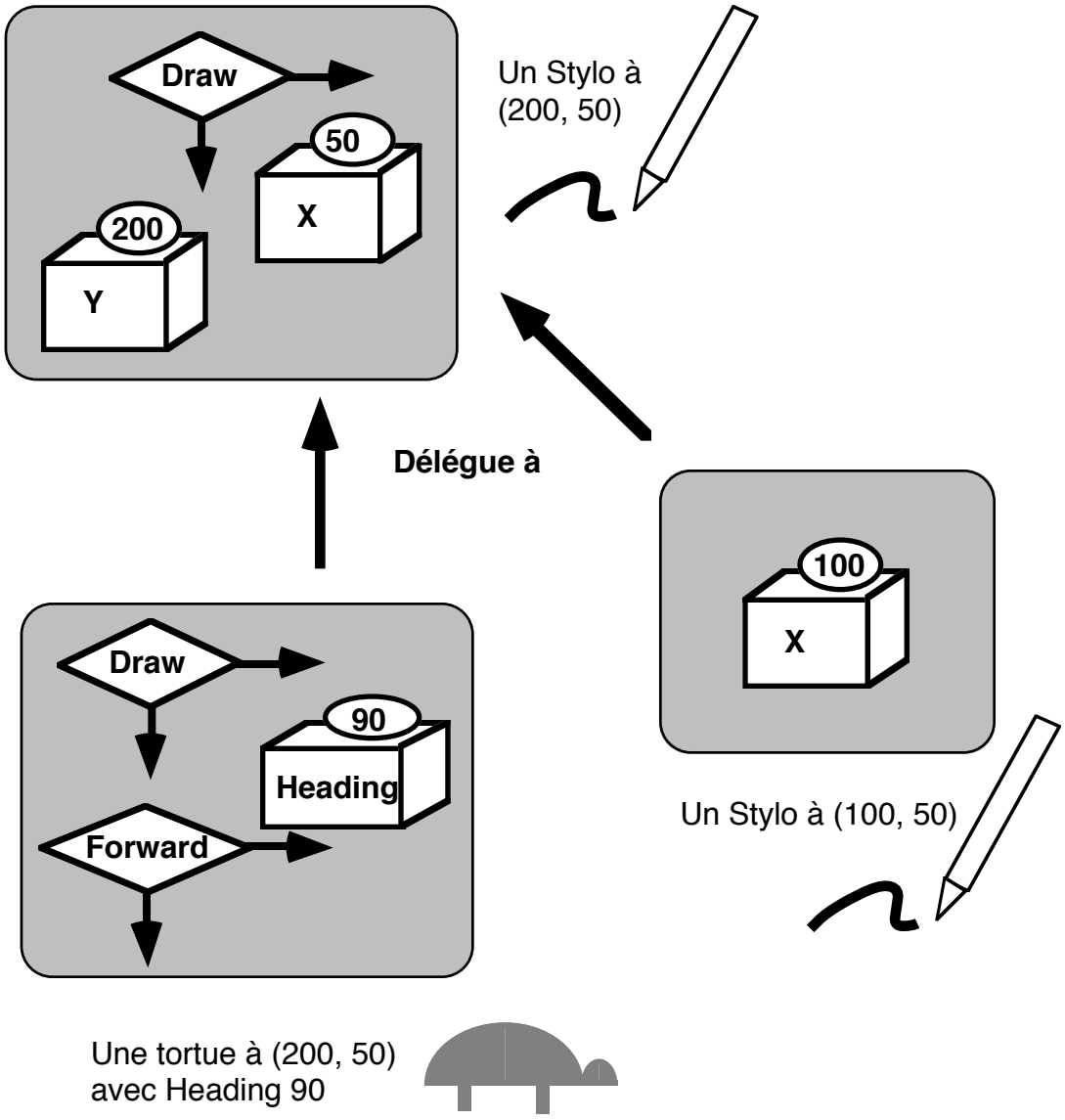
La technique de la délégation apparaissait déjà dans Act 1 [Lieberman 81 Preview], mais je n'ai pris conscience de son importance et de ses conséquences qu'après les 3èmes Journées d'études sur les langages à objets à Paris [Lieberman 86 Delegation] quand j'ai été invité par Pierre Cointe à les expliquer. Mon article d'OOPSLA 86 [Lieberman 86 Prototypes] a connu un grand succès, et depuis il a provoqué de nombreuses discussions publiées dans des conférences OOPSLA et ECOOP. Mes dernières réflexions sur le sujet se trouvent en [Stein, Lieberman, Ungar 89 Treaty].

Nous pouvons introduire la controverse à partir d'une question philosophique : comment représenter l'expression de relations de ressemblance entre objets ? La notion abstraite d'ensemble, définie par une énumération de ses membres ou par les critères d'appartenance est une façon de représenter cette abstraction. L'idée de classe est la réalisation de cette approche dans un langage à objets. La classe décrit le comportement commun de ses instances. Les instances sont des valeurs concrètes qui sont les éléments de l'ensemble. Cette idée est très utile en mathématique, mais elle pose des problèmes pour les langages à objets.

L'abstraction, qui est l'atout majeur de l'idée de classe, est parfois un inconvénient. J'ai observé, lorsque j'enseignais Logo, que c'est souvent la compréhension des exemples concrets qui précède la formulation des idées abstraites. On voulait identifier une situation concrète comme un *prototype*, et engendrer les conceptions plus abstraites par la généralisation progressive des aspects de la situation. J'ai également développé ce principe dans mes systèmes pour la programmation par l'exemple [Lieberman 86 Example], qui seront décrits plus loin dans ce document. L'idée de classe présuppose l'existence de la classe avant ses instances. Souvent, la rencontre d'une nouvelle situation exige la modification de la hiérarchie de classes.

Avec la délégation, un objet peut être décrit en exprimant ses différences par rapport à un prototype. Lorsque l'objet reçoit un message, il essaye d'y répondre en prenant en compte sa propre connaissance. Si ces différences avec le

prototype ne sont pas pertinentes, l'objet délègue le message au prototype afin de lui laisser y répondre. L'expédition de message au prototype au moment de la réception est une relation plus dynamique que celle entre une instance et sa classe.



Un hiérarchie d'objets qui partagent des connaissances grâce à la délégation. En haut, un prototype de stylo, et à droite, un autre stylo dont l'abscisse X est différente de celle du prototype. En bas, la tortue possède, en plus des connaissances propres à un stylo, la variable HEADING et la méthode FORWARD.

L'héritage fixe la relation de communication entre l'instance et la classe au moment de la création de l'instance. La délégation est donc plus adaptée aux situations dans lesquelles les relations de partage sont conditionnelles ou dynamiques. Je soutiens cette idée dans [Lieberman 86 Prototypes] grâce à un exemple de définition d'objets d'entrée-sortie qui combinent dynamiquement le

comportement de multiples sources. La délégation est également très adaptée à la représentation de propriétés par défaut [*defaults*].

Dans [Lieberman 86 Prototypes] je développe également des exemples montrant que la délégation permet de réduire souvent le nombre et la taille des objets, et simplifie leur représentation. J'y présente une comparaison de la puissance relative de la délégation et de l'héritage, et une analyse comparative des performances. J'y discute le problème technique lié à la pseudo-variable SELF des langages à classes, problème qui empêche d'implanter directement la délégation pour cette famille de langages. J'y traite finalement des conséquences relatives au parallélisme.

Le point de vue défendu dans [Lieberman 86 Prototypes] a favorisé la délégation d'une façon extrême, et cette position a été modérée en [Stein, Lieberman, Ungar 89 Treaty]. A l'occasion de OOPSLA-87 à Orlando (Etats-Unis), j'ai rencontré Stein et Ungar, qui avaient réalisé des systèmes (respectivement Hybrid et Self) ayant des mécanismes de partage à mi-chemin entre les classes traditionnelles et la délégation. Au lieu d'aboutir à un désaccord, la comparaison de nos systèmes respectifs nous a conduits à une nouvelle analyse de mécanismes de partage dans un langage à objets. D'où le "traité d'Orlando"□!

Nous y analysons plusieurs langages à objets à partir des notions du patron [*template*], et de l'empathie [*empathy*]. Un patron est un objet qui sert, comme une classe, comme modèle pour la création d'instances. L'empathie est une relation de similarité de comportement entre instances. Nous identifions trois dimensions indépendantes des choix des mécanismes de partage des langages à objets□:

• La relation de partage, doit-elle être statique ou dynamique?

• La spécification d'une relation de partage, doit-elle être implicite ou explicite?

• Le comportement d'un objet, doit-il être spécifié individuellement pour chaque objet ou collectivement pour un groupe d'objets?

A la limite, le choix d'une technique de partage dépend de l'évolution du programme. A la première époque de la vie d'un programme, il est souvent difficile d'anticiper les relations de partage entre objets. Le partage dynamique, explicite et individuel est donc conseillé. Plus tard, les changements du programme sont moins fréquents, et on supporte mieux le partage statique, implicite et collectif.

En dehors de mon travail sur les acteurs, j'ai travaillé aussi à des sujets d'intérêt général pour les environnements de programmation et pour l'intelligence artificielle. La prochaine partie du mémoire rappelle ma contribution à l'implantation des systèmes à objets, aux dépanneurs, à l'interface utilisateur et à la représentation de connaissances.

Le glanage de cellules

En général, je ne ne suis guère intéressé par les problèmes d'optimisation et les problèmes d'implantation de bas niveau, mais convaincu de l'existence d'un problème d'importance cruciale lié aux systèmes à objets, j'ai décidé de prêter une attention particulière au glanage de cellules. Souvent cité comme un obstacle majeur dans l'emploi de langages à mémoire dynamique à la Lisp ou Smalltalk, les algorithmes de glanage de cellules classiques exigeaient périodiquement une pause du calcul gênante. En proposant un glanage en temps réel, Henry Baker avait fait les premiers pas vers l'élimination de ce problème. Avec Carl Hewitt, nous avons découvert une idée simple qui bouleversait l'appréhension du problème du glanage de cellules : le glanage doit être réglé en fonction de la durée de vie des objets. Nous avons décrit un algorithme fondé sur cette idée dans [Lieberman, Hewitt 83 GC]. Depuis, cet article était référencé par pratiquement tous les articles traitant du sujet, et la plupart des réalisations contemporaines de langages à mémoire dynamique adoptent une variante de la stratégie qui y est décrit.

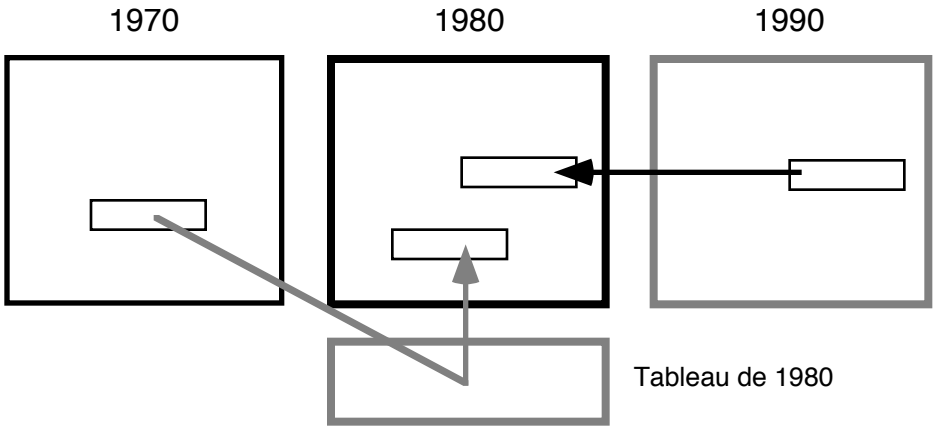
On peut observer que, pour un système standard, la plupart des objets ont une durée de vie relativement courte. Il est donc important d'optimiser le glanage de cellules en récupérant prioritairement les objets utilisés comme des mémoires temporaires, même si cette optimisation rend plus cher le coût des objets permanents. On peut associer cette stratégie à celle d'une simple "location" de mémoire, par comparaison à un "achat" de mémoire tel qu'il est pratiqué dans un système pour lequel le prix de l'allocation de mémoire est indépendant de la durée de vie.

La mémoire est divisée en régions appelées *générations*. La génération indique l'âge d'un objet. Nous distinguons les pointeurs en avant, i.e. ceux qui pointent d'un objet vers un autre plus jeune, et les pointeurs en arrière, ceux qui pointent d'un objet vers un autre plus âgé. Les opérations d'effets de bord, comme RPLACA en Lisp sont uniquement capable de produire les pointeurs en avant. Ceux-ci sont beaucoup plus rares que les pointeurs en arrière créés par les opérations de type CONS.

Notre algorithme copie incrémentalement les objets accessibles d'une région de mémoire vers une autre. Périodiquement, une région est condamnée [*condemned*]. Cette opération initialise la copie incrémentale de tous les objets qui s'y trouvent. L'opération de ramassage [*scavenging*], également incrémentale, supprime les références ayant la région condamnée comme destination. Les régions les plus jeunes sont condamnées très souvent, alors que les régions les plus vieilles ne sont condamnées que très rarement. Quand toute les références vers une région ont été copiées, la mémoire associée à cette région peut être récupérée. Le problème est alors de trouver toutes les références !

L'idée clé est celle-ci : parce que la plupart des pointeurs sont de type arrière, le gros du travail consiste à ramasser des générations plus jeunes que celle

condamnée. Notre algorithme exige donc un traitement spécialisé de pointeurs en avant. Pour ce faire, nous maintenons un tableau des pointeurs qui pointent en avant vers chaque génération. Quand une région est condamnée, il suffit de ramasser ce tableau et toutes les générations plus jeunes. Comme la plupart des régions condamnées correspondent aux jeunes générations, la mémoire occupée par les objets de durée de vie courte est récupérée très rapidement.



Pour récupérer la mémoire de la génération 1980, il suffit de copier les objets accessibles de cette région, et mettre à jour les références vers ces objets. Elles sont trouvables rapidement, parce que la plupart des références vers 1980 se trouvent dans les générations plus jeunes, i.e. 1990. Les rares pointeurs en provenance des vieilles générations sont enregistrés dans un tableau.

Nous discutons aussi dans [Lieberman, Hewitt 83 GC] d'autres questions pertinentes posées par le glanage de cellules. Nous suggérons des mesures d'efficacité pour un glaneur de cellules qui sont plus adaptées aux besoins des systèmes modernes que les mesures classiques. Nous discutons des éléments d'un langage de programmation de nature à poser des problèmes à un glaneur de cellules : la pile, les cellules de valeur, les "pointeurs faibles", le parallélisme. Nous concluons par l'observation qu'un style de programmation est très dépendant du prix relatif de la création des objets et des effets de bord. Nous sommes convaincus que la réduction du coût des objets à durée de vie courte facilite le style de programmation fonctionnel.

Les dépanneurs et la visualisation des programmes

A mon avis, l'augmentation de la productivité des programmeurs passe avant tout par l'amélioration des outils de dépannage. Les sondages montrent que les programmeurs consacrent la moitié de leur temps à l'activité de dépannage. Curieusement, peu de recherches sur le sujet ont été menées, tant en milieu académique qu'en milieu industriel. En fait, la technologie des dépanneurs n'a guère progressée pendant ces vingt dernières années.

Pourquoi un tel manque d'attention pour un problème aussi important ? Bien sûr, les programmeurs ont souvent honte de la présence d'erreurs dans leurs programmes. Ils hésitent à consacrer explicitement beaucoup d'efforts à améliorer les performances d'une activité qu'ils préfèrent minimiser. Ou peut-être sont-ils tellement séduits par les promesses de Dijkstra et ceux des tenants du génie logiciel, qui cherchent à éliminer complètement le besoin de dépannage.

J'ai pris très tôt conscience de l'importance de l'activité de dépannage du fait de mon enseignement de Logo. A la différence des cours d'informatique classiques, nous avons enseigné explicitement les techniques de dépannage. Nous avons encouragé les étudiants à considérer les erreurs comme des occasions d'approfondir leur compréhension du déroulement des programmes et de la représentation de problèmes.

L'intelligence artificielle et l'infographie devraient beaucoup contribuer à transformer les outils de dépannage. L'intelligence artificielle a appris à maîtriser les principes de résolution heuristique de problèmes et pourrait les appliquer au domaine de dépannage. Très peu d'environnements de programmation actuels tirent suffisamment bénéfice de la capacité graphique d'un écran haute résolution, de la couleur, de la typographie et de l'animation des nouvelles stations de travail.

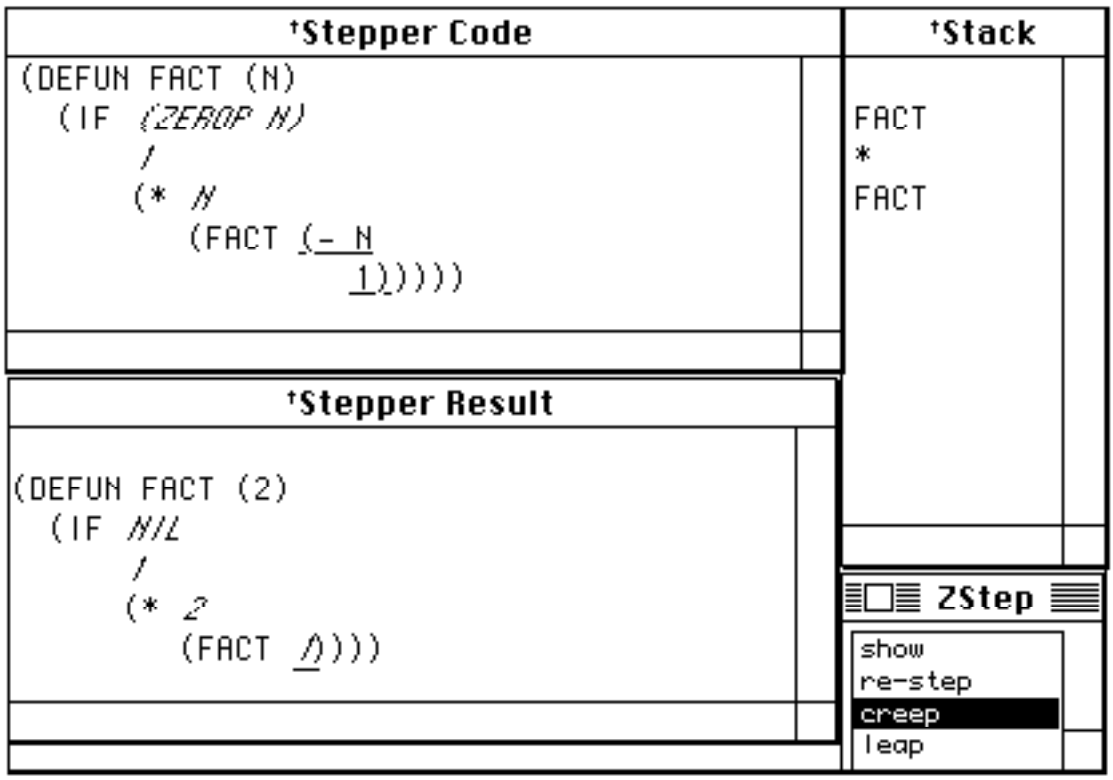
J'ai réalisé une série de dépanneurs graphiques expérimentaux pour Lisp. Le premier, pour la Symbolics, est nommé Zstep [Lieberman 84 Steps]. Une version améliorée pour le Macintosh est présentée dans [Lieberman 87 Reversible], où je décrit également les techniques de son implantation. La version la plus récente et la plus novatrice, emploie le graphique tridimensionnel. Elle est décrit dans [Lieberman 89 3D].

Dans [Lieberman 89 CHI], j'ai analysé le problème du dépannage. J'ai fait un bilan des recherches antérieures et une bibliographie exhaustive sur le sujet. Au Visible Language Workshop, nous travaillons sur un projet appelé "Le graphique pour la visualisation des programmes", subventionné par Hewlett-Packard, consacré à l'investigation des nouvelles représentations visuelles pour le dépannage et donc pour la programmation.

Le dépanneur ZStep est un dépanneur pas-à-pas. Les dépanneurs pas-à-pas sont en général les meilleurs outils pour la localisation des erreurs, mais la structure de contrôle des dépanneurs est souvent mal adaptée aux besoins du processus de dépannage. ZStep est conçu pour soutenir les processus heuristiques de diagnostic des causes d'une erreur. Il donne au programmeur une vue d'ensemble du programme, suivi par une succession de vues de plus en plus raffinées jusqu'à la découverte de l'erreur. Le programmeur peut exécuter un programme jusqu'à l'occurrence d'une erreur ou jusqu'à l'obtention d'une valeur inattendue, et relancer alors le programme en arrière pour remonter à la cause de l'erreur. ZStep mémorise pour cela l'histoire complète du calcul.

ZStep anime dynamiquement l'exécution d'un programme par la substitution visuelle d'expressions. Le code du programme est affiché sur l'écran, et à chaque pas, la prochaine expression à évaluer est soulignée. Après chaque pas de son évaluation, la valeur obtenue remplace sur l'écran le code de cette expression. Cette technique est simple et intuitive, et elle permet de visualiser en permanence de quel contexte est issu une expression.

ZStep dispose aussi d'un mode employant deux fenêtres, une pour le code original, l'autre pour le code avec les valeurs substituées. Ce mode met bien en évidence la correspondance entre une expression et sa valeur. Une version pour le Macintosh, réalisée avec la collaboration de Christopher Fry, propose une troisième fenêtre pour visualiser dynamiquement la pile.



Zstep propose une approche inédite du traitement des erreurs. Dans les systèmes classiques, l'occurrence d'une erreur arrête le programme. Le programmeur doit choisir entre lancer immédiatement un dépanneur au point d'arrêt [*breakpoint*] pour, par exemple, examiner la pile, ou entre retourner immédiatement à la boucle d'interaction principale, en perdant toute l'histoire du calcul.

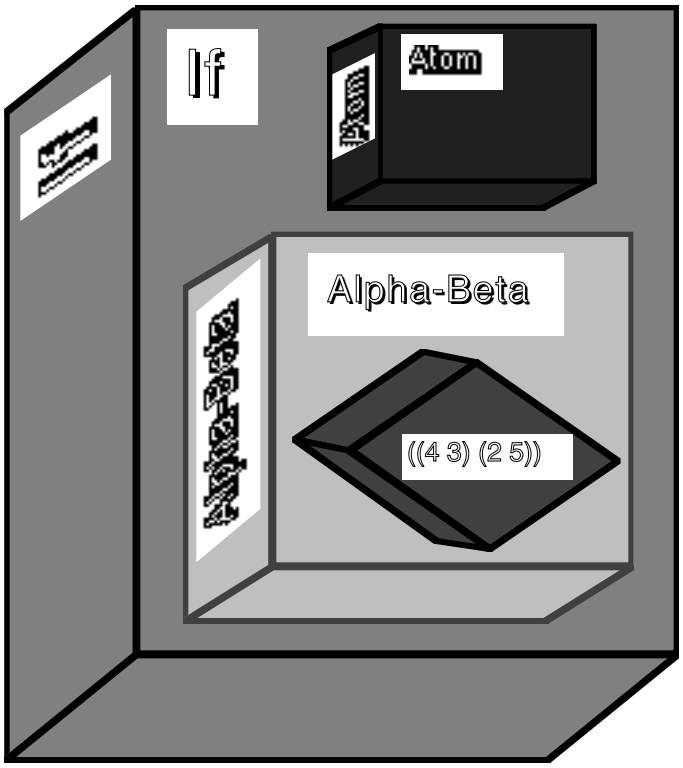
Zstep, en revanche, crée des objets erreurs, qui remplacent la valeur retournée par une expression normale. L'expression est remplacée par l'affichage du message d'erreur sur l'écran. Le programme n'est pas interrompu, et aucun dépanneur de

point d'arrêt n'est nécessaire. Le programmeur peut continuer à faire fonctionner le programme, en avant ou en arrière. Il peut se livrer à autres activités, et revenir plus tard pour chercher les causes de l'erreur.

Au Visible Language Workshop, je continue mon travail sur les représentations graphiques pour le dépannage, en collaboration avec des concepteurs graphistes. Pour la représentation textuelle du programme, les propriétés typographiques sont employées pour mettre en évidence les sous-expressions et établir la correspondance entre des expressions reliées entre elles, qui sont néanmoins séparées sur l'écran. Une version plus récente pour la Symbolics remplace la représentation du texte du programme par une représentation graphique, basée sur un arbre de nœuds et de liens montrant le graphe des appels fonctionnels. Le même processus dynamique de substitution visuelle s'applique également à la représentation graphique. Les représentations multiples d'un programme sont animées simultanément.

Avec l'arrivée des nouvelles stations de travail, l'emploi de la couleur introduit une nouvelle dimension. Nous employons la couleur de deux façons distinctes. Nous choisissons une palette de couleurs pour exprimer le statut temporel des expressions : une couleur pour les expressions non évaluées, une autre pour les expressions en cours d'évaluation, une autre encore pour la valeur retournée. Nous employons aussi un spectre de couleurs continu pour afficher d'autres effets visuels. Les effets de transparence et de translucidité réalisent les transitions imperceptibles entre deux états successifs.

Les stations de travail munies de processeurs pour l'accélération de l'affichage des images en trois dimensions donnent la possibilité d'utiliser ces images pour la programmation. Dans [Lieberman 89 3D] je présente une représentation iconique abstraite d'un programme, animé en trois dimensions sur une station de travail HP. La représentation emploie des boîtes colorées pour représenter les fonctions, et la relation fonction-arguments y est traduite par la relation d'inclusion des boîtes. La troisième dimension est employée pour afficher sur les côtés des boîtes une vue de la pile. L'intégration de la vue de la pile à celle du code évite la séparation spatiale de ces deux vues caractéristiques des autres dépanneurs.



Une représentation tridimensionnelle de l'exécution d'un programme. Les boîtes représentent les appels de fonctions, et la relation d'inclusion modélise la relation fonction-argument. Le losange représente une valeur.

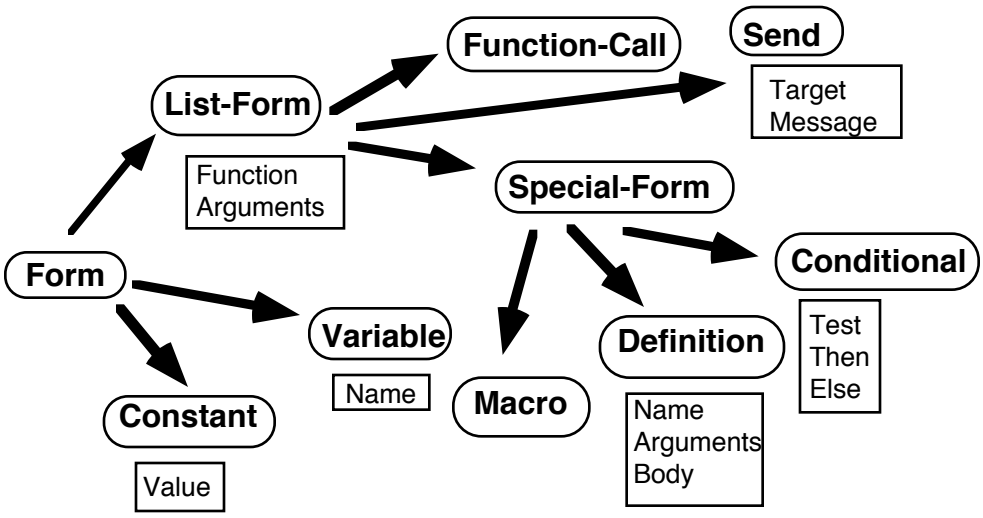
L'animation se déroule par un processus de changement dynamique de la racine de l'arbre d'inclusion des icônes. Ce processus de changement de la racine, est appelé *marcottage*. Nous l'expliquons et le perfectionnons dans [Lieberman 90 Marcottage]. Les boîtes grandissent, rétrécissent, pivotent, et changent de couleur en fonction des opérations exécutées par le programme.

La correspondance entre l'état du programme et les propriétés graphiques de l'image constitue une langue visuelle de l'exécution dynamique d'un programme.

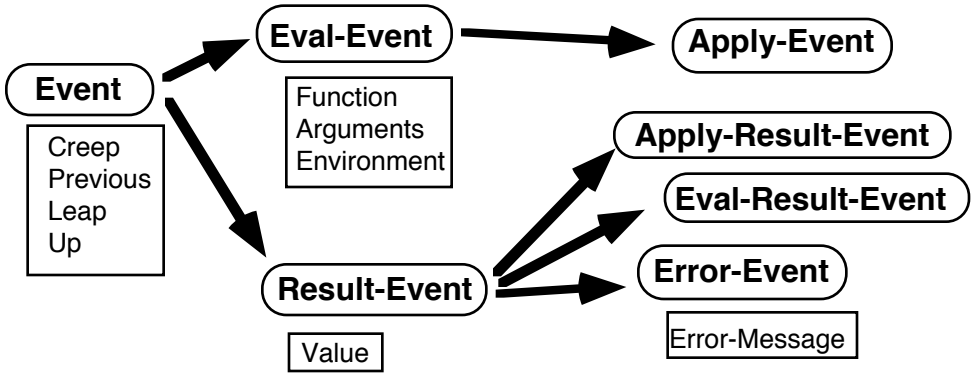
Au-delà de l'emploi du graphisme pour la visualisation des programmes pendant le dépannage, une possibilité plus audacieuse est l'emploi d'illustrations comme support unique à la programmation. Le texte du langage de programmation serait purement et simplement supprimé. Cette idée est séduisante, surtout pour attirer à la programmation des "penseurs visuels" comme les concepteurs graphistes, les artistes, ou les architectes. Les premières expériences dans ce domaine datent de l'apparition des premiers écrans graphiques, mais beaucoup de chercheurs restent sceptiques quant à la mise en œuvre de cette idée. Dans [Lieberman 89 Visual Programming] j'analyse le pour et le contre de cette idée, et je conclus que l'amélioration rapide des capacités graphiques des stations de travail représente une bonne opportunité de faire avancer l'idée de "programmation visuelle".

Les évaluateurs réversibles

Pour accéder à sa structure de contrôle, le dépanneur pas-à-pas dispose d'un évaluateur réversible, exposé dans [Lieberman 87 Reversible]. Chaque programme est représenté d'une manière objet, avec une classe d'objets pour chaque élément du langage Lisp : les variables, les appels de fonctions, les conditionnelles. Nous utilisons aussi une représentation objet des états dynamiques de l'exécution du programme, composée d'objets nommé "événements". Les événements sont liés bidirectionellement, par ordre chronologique et par leur position dans l'arbre d'évaluation.



La hiérarchie statique de programmes



La hiérarchie dynamique de programmes

Pendant mon travail sur les dépanneurs, j'ai rencontré de graves problèmes pour déterminer précisément le comportement de l'évaluateur Common Lisp, afin de modéliser l'évaluateur dans mon interprète objet. Ce problème est d'autant plus crucial que l'on travaille avec des réalisations de Common Lisp pour différentes

machines. Malgré la normalisation de Common Lisp, le comportement de l'évaluateur reste mal spécifié. Avec Christopher Fry, nous avons proposé au comité Common Lisp de normaliser le comportement de l'interprète par le biais d'un programme méta-circulaire écrit en Common Lisp, appelé *Common EVAL* [Lieberman, Fry 88 Common EVAL]. Une telle normalisation est indispensable si l'on désire écrire des dépanneurs portables et des aides à l'environnement de programmation qui soient portables.

Nous voulions intégrer le processus d'édition du programme à celui de dépannage. Avec un dépanneur pas-à-pas, le programmeur examine l'évaluation jusqu'à la découverte de l'erreur. Pendant cette recherche, le programmeur devrait pouvoir éditer le programme, puis reprendre le calcul. Les environnements interactifs Lisp et Smalltalk permet une telle édition, mais la granularité est celle d'une fonction ou méthode. Est-il possible d'éditer une seule expression d'un programme qui est déjà en cours d'évaluation□?

J'ai récemment réalisé des progrès récents dans ce domaine à l'aide d'un évaluateur objet basé sur la technique du *marcottage* [Lieberman 90 Marcottage]. Le marcottage est une technique de navigation permettant de parcourir incrémentalement une structure, comme l'évaluateur classique parcourt un programme, d'effectuer des "éditions virtuelles" de la structure, puis de reprendre le parcours. J'explique ci-dessous cette technique de marcottage.

Les interfaces utilisateur

Les interfaces graphiques, avec menus, icônes et objets graphiques manipulables à la souris, ont révolutionné l'interaction entre l'utilisateur et l'ordinateur. Mais la programmation de ces interfaces reste compliquée et truffée de pièges. La gestion des clics de souris, des menus, des fenêtres et la correspondance entre les gestes de l'utilisateur et les objets graphiques n'ont rien d'évident.

A mon avis, les interfaces graphiques sont les "langages de programmation" de l'avenir. Avec les micro-ordinateurs, les utilisateurs peuvent souvent se servir d'une application de l'interface graphique au lieu d'écrire un programme dans un "vrai" langage de programmation.

Mais qu'est-ce qu'un langage de programmation□? Au fond, un langage de programmation est un langage, traditionnellement textuel et symbolique, avec lequel l'utilisateur apprend à l'ordinateur à réaliser ses intentions. Une interface graphique est également un langage qui permet à l'utilisateur de demander à l'ordinateur de réaliser ses intentions, mais il s'agit d'un langage visuel et démonstratif.

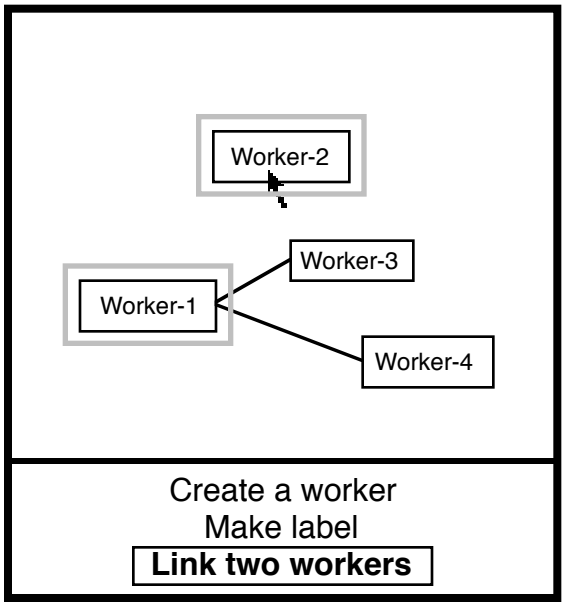
Si l'on prend au sérieux l'analogie entre les interfaces graphiques et les langages de programmation, on peut se poser la question suivante□: existe-t-il un interprète pour une interface graphique□? Malheureusement la réponse est le plus souvent□:

il n'y en a pas. On est donc obligé de réécrire un interprète, avec la boucle d'interaction principale, pour chaque nouvelle application. Imaginons que chaque programme Lisp soit obligé d'écrire répétitivement son propre interprète Lisp□!

Un interprète pour un langage de programmation comme Lisp fournit une fonction de lecture, capable de traduire l'entrée, le texte du programme, sous forme d'objets Lisp. L'interprète détermine à quelles fonctions l'utilisateur veut faire appel, collecte les arguments des fonctions, calcule les résultats et les affiche. Un interprète adapté à une interface graphique doit traduire l'entrée (ici, les déplacements et les clics de souris) aux actions de l'interface□: appel aux opérations des menus, déplacements d'objets graphiques. L'interprète associe les sélections de menu ou d'icônes à des fonctions de l'application, collecte des arguments spécifiés par la sélection à la souris, les boutons, ou les boîtes de dialogue. Il calcule les résultats et réaffiche l'écran.

Dans mon article de SigGraph 85 [Lieberman 85 Menus], je présente Ezwin, un interprète pour un langage des opérations d'interface graphique. C'est un langage dans lequel les sélections de menus et les opérations iconiques sont les fonctions, et les objets graphiques sensibles à la souris sont les arguments. L'interprète réalise une boucle d'interaction principale commune à toutes les interfaces graphiques. Outre la sélection de fonctions de l'application par menu ou icône, et la sélection d'arguments par la désignation à la souris de leurs représentations graphiques, l'interprète rend des services comme la gestion des erreurs, l'annulation de commandes, et l'aide à l'utilisateur.

L'interprète est réalisé d'une manière orientée objet. Il existe trois catégories principales d'objets□: des objets qui représentent l'application dans son intégralité, un objet pour chaque commande comprise par l'application, et un objet pour chaque représentation graphique sur l'écran; ces derniers sont appelés les *présentations*. L'interprète est donc un éditeur pour les présentations.




```
(Link-two-workers Worker-1
                  Worker-2)
```

Une interface graphique au moment où le deuxième argument de la commande LINK-TWO-WORKERS est sélectionné. L'interprète Ezwin met automatiquement en évidence la commande de menu et l'argument WORKER-1 déjà choisis. Pendant la sélection du deuxième argument, seuls les objets de type WORKER sont sensibles à la souris. La fonction ne doit déclarer que le nombre et les types des arguments.

Chaque action de l'interface, comme mise en évidence visuelle d'un objet graphique sélectionné, est traduite par un message accepté par l'objet, avec un comportement par défaut. Si le programmeur désire changer le comportement d'objets sélectionnés, il lui suffit d'ajouter de nouvelles méthodes qui modifient ce comportement. Il est facile de concevoir des objets qui représentent les styles d'interaction. On peut également définir des objets représentant des commandes génériques, comme les "couper-coller".

Afin de démontrer la puissance de cette approche, j'ai réalisé un interprète d'interfaces graphiques doté de quelques innovations. Beaucoup d'interfaces font le choix entre deux processus de sélection de fonctions et d'arguments. Il existe des interfaces pour lesquelles on choisit d'abord une fonction, et ensuite, ses arguments. Il existe d'autres interfaces pour lesquelles la sélection des arguments précède la sélection de la fonction. Pourquoi ne pas permettre les deux approches à la fois□?

J'ai réalisé un tel interprète dans lequel la sélection des arguments peut précéder ou suivre la sélection de la fonction, au gré de l'utilisateur, et sans aucune prévision au niveau de l'application. De plus, la sélection est toujours sensible au contexte. Si l'on sélectionne une fonction, seuls les objets qui sont candidats à être l'argument suivant sont accessibles. Si l'on sélectionne une séquence d'arguments, seules les fonctions qui accepteraient ces arguments sont sensibles à la souris. L'objet représentant la fonction fait une déclaration du nombre et du type des arguments désirés, et la responsibilité de l'interprète est de faire la

correspondance entre les objets sélectionnés et les fonctions et les arguments disponibles. L'interface gagne en souplesse et la possibilité d'erreur est réduite.

[Lieberman 86 Debugging] traite conjointement de mes préoccupations relatives aux interfaces utilisateur et de celles concernant le dépannage, en abordant le problème du dépannage des interfaces utilisateur, trop souvent ignoré. Beaucoup d'outils de dépannage classiques textuels, comme des dépanneurs pas-à-pas classiques, les points d'arrêt [breakpoints] ou les traceurs deviennent inutiles en présence de sorties multi-fenêtrées, d'entrées graphiques et du parallélisme, qui caractérisent les nouvelles interfaces utilisateur graphiques. Comment adapter ces outils pour répondre aux besoins des interfaces modernes□?

Je continue aujourd'hui ce travail sur les interfaces utilisateur. Pendant mon séjour à Paris, j'ai travaillé sur un système d'interface utilisateur, successeur de Ezwin, appelé Suit. Un problème avec les systèmes d'interface classiques est l'absence de "récursivité". En général, on peut construire une interface à partir des composants primitifs, comme les fenêtres, les boutons, ou les menus, mais l'interface qui résulte ne peut pas être un composant d'une interface plus grande, construite de la même façon.

Suit est basé sur un objet appelé interacteur [*interactor*], qui dénote en même temps les notions d'"interaction" et d'"acteur". Un interacteur représente une interface utilisateur, avec ses objets affichés et son savoir-faire décrivant ses réponses aux gestes de l'utilisateur. On peut fabriquer des interfaces complexes à partir d'interacteurs primitifs, et le résultat est à son tour un interacteur, puisqu'on peut composer les acteurs à partir d'acteurs plus simples. Suit permet aussi de définir une interface en suivant les déplacements de la souris [*mouse tracker*] par l'intermédiaire d'une boucle d'interaction principale prédéfinie.

L'art et l'informatique

Je m'intéresse depuis longtemps à l'application de l'informatique aux domaines artistiques. Depuis mon travail graphique pour Logo, je me suis toujours interessé à l'utilisation de l'ordinateur comme un outil artistique. J'ai le bonheur de travailler actuellement au Media Laboratory, où beaucoup de créateurs des milieux artistiques collaborent avec les informaticiens. A mon avis, l'intelligence artificielle aurait tort d'ignorer la partie artistique de l'intelligence humaine.

J'ai réalisé par ordinateur des œuvres d'art abstraites, qui le plus souvent faisaient appel à la technique des objets fractals. Les fractals ont toujours joué un rôle dans l'enseignement de Logo, et j'ai été très tôt inspiré par Benoit Mandelbrot, que j'ai rencontré dès 1976, bien avant la parution de ses livres en anglais. En 1976, j'ai obtenu le premier prix au premier concours d'art par ordinateur de la revue *Byte*. En 1982, j'ai exposé une image à l'exposition d'art de la conférence américaine d'infographie SigGraph. J'ai également exposé des photographies dans des galeries d'art.

J'ai réalisé des illustrations pour des couvertures de livres [le livre *Economics* du lauréat du prix Nobel Paul Samuelson, un livre scolaire de mathématique, un calendrier d'art par ordinateur], et de disques [un disque de jazz, et plus récemment, deux disques compacts de musique par ordinateur]. J'ai participé à plusieurs colloques "New Graphics" animé par le MIT Visible Language Workshop [avant son intégration au Media Laboratory]. J'ai collaboré avec des musiciens et des danseuses, notamment à la série de concerts et représentations du New England Computer Arts Association à Boston.

Je m'intéresse aussi à la musique. Pour le *Computer Music Journal*, j'ai écrit un article sur l'application des technologies de programmation par objets, de l'intelligence artificielle et des interfaces à la musique par ordinateur [Lieberman 82 Music], [Lieberman 87 Synthesizer]. Enfin, l'article [Lieberman 88 Fifth-Generation] s'applique avec pertinence au domaine musical.

La programmation par l'exemple

A l'époque du projet Logo, les enseignants ont introduit l'idée de programmation aux étudiants par l'intermédiaire d'une métaphore : la programmation est comme l'enseignement. Le programmeur est le professeur, l'ordinateur est l'étudiant. Le but du professeur est d'enseigner un processus à l'étudiant, pour que l'étudiant puisse ensuite exécuter tout seul ce processus. Malheureusement, l'ordinateur est un étudiant parfaitement stupide, et il faut expliciter le processus très exactement, par une séquence de commandes très détaillées. Ces commandes constituent le programme.

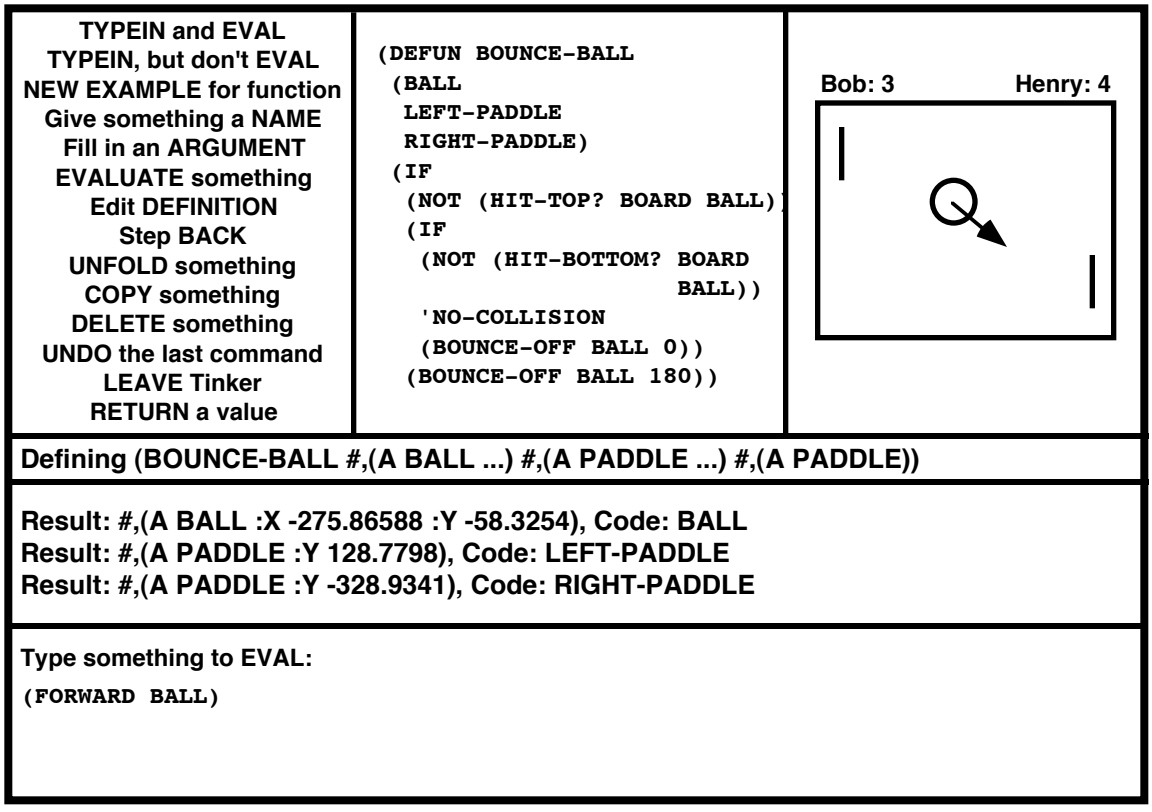
Cette métaphore a été une grande réussite pour enseigner la programmation. L'idée de la programmation comme une activité d'enseignement est présente aussi en intelligence artificielle. Le but de l'apprentissage automatique [*machine learning*] est de remplacer la programmation conventionnelle par un processus qui ressemble à l'enseignement humain.

Mais j'ai très vite perçu une grande différence entre ma propre expérience de l'enseignement et mon activité de programmation : le rôle des exemples. Un bon professeur emploie toujours des exemples pour illustrer ses explications. Un bon étudiant vérifie toujours sa compréhension d'une idée en faisant l'effort de l'appliquer à un exemple concret.

Psychologiquement, les êtres humains sont plus adaptés à se rappeler de situations concrètes, puis à les généraliser afin de former des idées abstraites, plutôt que de procéder de façon inverse. Les exemples servent à concrétiser les étapes intermédiaires d'un processus. En l'absence d'exemples, l'étudiant est obligé d'imaginer les conséquences concrètes d'une idée abstraite, ce qui met à rude épreuve sa mémoire à court terme.

Je me suis alors posé la question suivante : pourquoi ne pas employer les exemples pour enseigner un processus à l'ordinateur dans un environnement de programmation ? Cette question m'a conduit à travailler sur le sujet de la programmation par l'exemple [*programming by example*]. J'ai donc réalisé un système expérimental qui s'appelle *Tinker*, support à la programmation par l'exemple en Lisp.

Tinker est un environnement de programmation dans lequel on peut définir une fonction Lisp à partir d'une démonstration d'un calcul sur des exemples concrets. Au début, on choisit des exemples de données d'entrées typiques pour la fonction désirée. Pour chaque ensemble d'arguments, on montre à Tinker les étapes d'évaluation de la fonction, en employant les valeurs immédiates d'arguments. Tinker affiche le résultat de chaque étape, et il enregistre une version généralisée de l'étape. La réaction immédiate donne à l'utilisateur la possibilité de vérifier si le résultat de l'étape est bien celui désiré. Dans la négative, l'utilisateur peut effectuer l'édition tout de suite. La vérification du programme sur des exemples concrets est donc intégrée au processus d'écriture du programme.



Un écran de Tinker. En haut, le menu, le code du programme synthétisé, et la visualisation graphique du programme en exécution [un jeu de tennis]. Au milieu, des expressions du code et les valeurs associés. En bas, la fenêtre d'interaction textuelle.

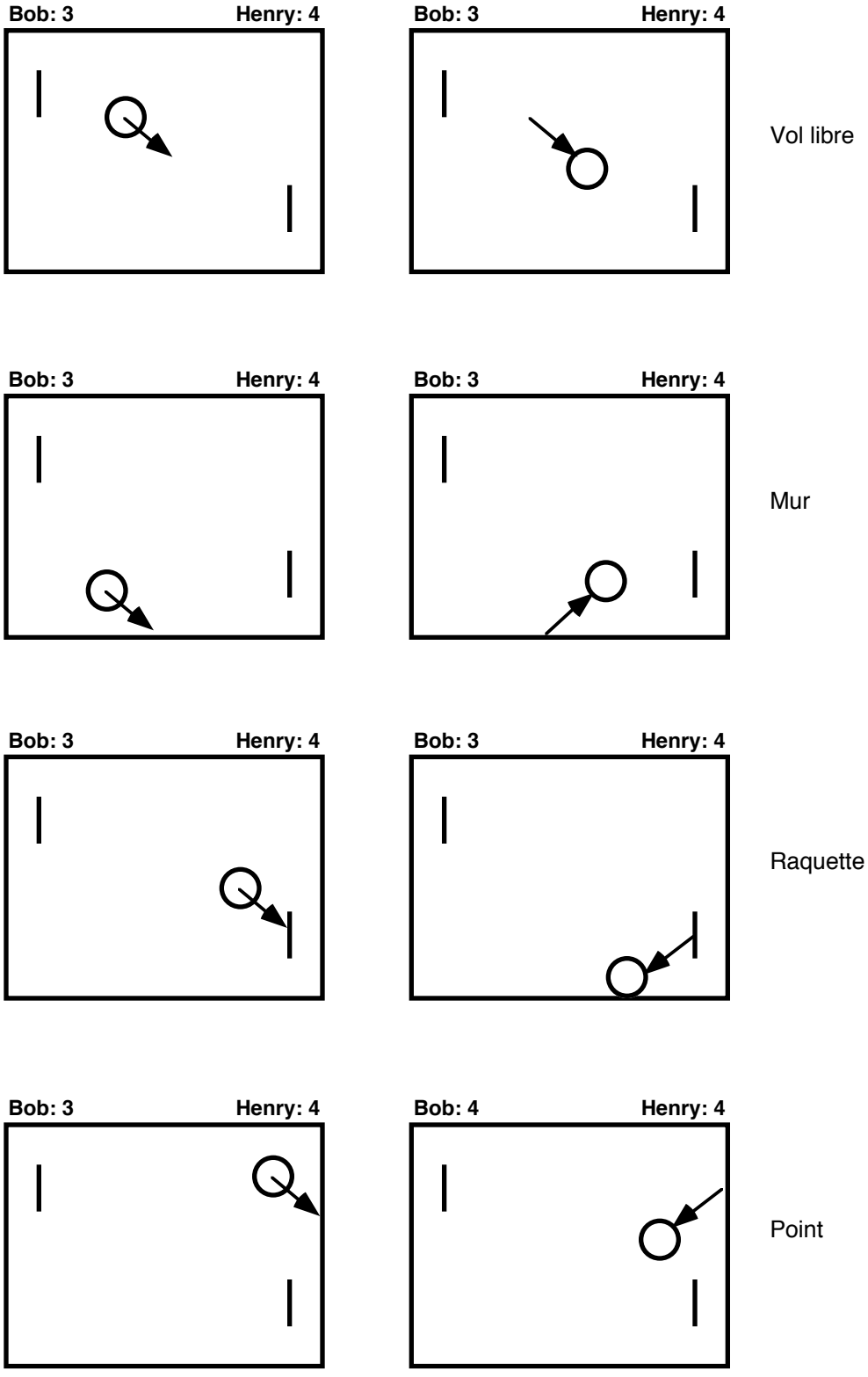
Pour la généralisation d'un programme à partir d'exemples, un système pour la programmation par l'exemple demande à connaître quelles parties du

programme pourront changer à chaque nouvelle invocation de la fonction. Il existe des systèmes qui essayent de faire ce choix automatiquement, mais si l'on veut préserver la capacité de synthétiser n'importe quel programme, l'avis de l'utilisateur relatif à cette question est indispensable. Pour Tinker, ce sont les variables d'arguments désignées par l'utilisateur qui indiquent la généralisation. Tinker maintient toujours les contraintes de dépendance entre les expressions pour diffuser la généralisation. Si l'on appelle une fonction sur un argument généralisé, le résultat est aussi généralisé. Cette gestion de dépendances est effectuée par un interprète Lisp réversible, décrit dans [Lieberman 87 Reversible].

Tinker dispose de la capacité de généraliser à partir d'exemples multiples. Dans l'enseignement, un professeur emploie souvent une séquence d'exemples liés, soigneusement choisis, pour démontrer à l'étudiant une idée complexe. Chaque exemple dépend de la compréhension des exemples précédents, et chaque exemple introduit un nouvel aspect de l'idée. Cette capacité est essentielle pour l'introduction des fonctions conditionnelles et récursives.

Lorsque Tinker rencontre des exemples multiples pour la même fonction, il pose une question à l'utilisateur□: quelle est la différence entre ces exemples□? C'est à l'utilisateur de fournir le critère de différenciation. Tinker a l'intelligence de choisir l'exemple le plus proche parmi ceux déjà présentés pour servir comme exemple pour montrer la discrimination. Les exemples sont comparés à partir du code associé, et Tinker choisit l'expression minimale de différence entre les exemples pour synthétiser la conditionnelle. Tinker vérifie que les critères donnés parviennent en fait à distinguer les exemples.

Pour un processus conditionnel, l'utilisateur démontre un exemple pour chaque partie, la vraie et la fausse, de la conditionnelle. Pour définir une fonction récursive, il faut présenter un exemple pour le cas fondamental, et ensuite, un exemple pour le cas récursif, qui se réduit finalement au cas fondamental. Lorsque Tinker doit générer une expression conditionnelle, il est nécessaire que l'utilisateur donne un exemple de ce que le futur programme doit faire lorsque le test est vrai et lorsqu'il est faux. Dans le cas d'une fonction récursive, l'utilisateur doit donner un exemple d'arrêt de la récursion [(FACT 0) = 1] et un exemple de la formule générale de récursion [(FACT 7) fait appel à (FACT 6)]. Comme Tinker peut générer les variables, les conditionnelles et les fonctions récursives, il est capable de synthétiser n'importe quel programme Lisp.



Les exemples du déroulement d'une partie de tennis. Chaque exemple est composé de deux écrans qui montrent l'état du jeu avant et après une transition.

L'approche de l'apprentissage de Tinker ressemble à celle appelée plus tard par Mitchell, la généralisation à partir d'explications [*explanation-based generalization*].

Mais ici, l'ordre de présentation est différent, car l'explication est présentée incrémentalement afin d'arriver à la généralisation désirée. On peut designer cette approche par l'expression : "l'explication à partir de généralisation".

Il est important de ne pas confondre la programmation par l'exemple, telle qu'elle apparaît ici, avec d'autres recherches basées sur l'apprentissage par l'exemple, comme la recherche connexioniste qui est aujourd'hui à la mode, ou celle que j'appelle la programmation par l'exemple de l'histoire des entrées-sorties [*programming by examples of input-output histories*]. Cette dernière recherche s'efforce de généraliser un programme à partir d'une présentation de l'entrée et de la sortie de la fonction désirée. Par exemple, d'après l'entrée (A B C) et la sortie (C B A) on pourrait synthétiser la définition habituelle de REVERSE, sans aucune indication sur le processus intermédiaire. Mais imaginons que l'on veuille apprendre le jeu d'échecs à partir de la seule configuration initiale de l'échiquier et des exemples de configurations d'échec et mat !

Les domaines qui peuvent bénéficier de la programmation par l'exemple sont nombreux. En éducation, la programmation par l'exemple serait une excellente technique d'introduction à la programmation. Les débutants ont tendance à penser d'une manière plus concrète que les experts. La programmation par l'exemple est particulièrement efficace dans les situations où la distance entre la compétence actuelle du programmeur et la difficulté du problème à résoudre est grande. Cette distance est grande pour le débutant, parce qu'il n'a pas encore beaucoup d'expertise. En intelligence artificielle, cette distance est également grande, mais pour la raison inverse : la difficulté du problème. La difficulté des problèmes en intelligence artificielle rend impossible une analyse algorithmique complète, et la seule approche possible est d'employer des techniques heuristiques qui fonctionnent sur des exemples particuliers.

Pour les interfaces graphiques interactives, un exemple est un "scénario" d'interaction typique d'un utilisateur avec l'interface. On trouve de tels scénarios dans les manuels d'introduction, ou pendant la planification d'un projet. Le scénario est composé de séquences d'exemples d'entrée d'utilisateur (clics de la souris, sélections de menus), et d'exemples de sorties graphiques (objets graphiques réaffichés). La programmation par l'exemple permet de créer le code de l'interface à partir du scénario. Une fenêtre de l'écran est consacrée au "point de vue de l'utilisateur", où l'interface est affichée telle que l'utilisateur l'aurait vu. Le programmeur peut vérifier incrémentalement la qualité de l'interaction pendant l'écriture du programme. Je présente de nombreux exemples de ce style de génération d'interfaces dans mes travaux cités ci-dessous.

Naturellement, la meilleure façon d'expliquer la programmation par l'exemple est par l'exemple. La plupart de mes articles sur la programmation par l'exemple sont consacrés à la présentation d'exemples d'emploi de cette technique dans des domaines divers. J'ai décris la première version de Tinker dans [Lieberman, Hewitt 80 Tinker]. Cet article explique le mécanisme de base de la généralisation de programmes à partir d'exemples, et le mécanisme d'enregistrement de dépendances entre expressions. Nous présentons des exemples de fonctions récursives simples en Lisp, telle que DERIVATIVE.

J'ai présenté Tinker à la communauté de l'intelligence artificielle à l'occasion d'IJCAI-81 à Vancouver [Lieberman 81 Tinker]. Le domaine des interfaces graphiques est fécond pour la programmation par l'exemple, et j'ai présenté l'exemple d'un tableur dans [Lieberman 82 Interfaces], et celui d'un éditeur de menus dans [Lieberman 82 Interactive]. Les connexions entre mon travail et le monde de l'éducation sont développées dans [Lieberman 86 Example].

L'utilisation de la programmation par l'exemple pour la visualisation graphique d'algorithmes est présentée dans un article plus long paru dans le journal Man-Machine Studies [Lieberman 84 Seeing]. J'y emploie à titre d'exemple un algorithme de recherche alpha-béta, très connu en intelligence artificielle.

J'ai enregistré une démonstration vidéo, diffusée au grand public par la série SigGraph Video Review [Lieberman 85 Example Video]. Ce film montre la programmation d'un jeu de tennis animé. On présente les exemples de la position de la balle et des raquettes, et l'on explique au système les règles décrivant le comportement dynamique de la balle et exprimant dans quelles conditions un joueur peut gagner un point.

En intelligence artificielle, j'ai étudié une application de la méthodologie de programmation par l'exemple à un langage de représentation de connaissances, pour décrire les analogies [Lieberman 89 Analogies]. On peut voir une analogie comme la construction d'une description abstraite d'un exemple concret qui peut être appliquée à un autre exemple. Dans cet article, je propose un langage graphique pour communiquer les analogies incrémentalement à partir d'exemples.

Au Media Laboratory, j'ai travaillé sur des systèmes pour la programmation par l'exemple appliqués aux domaines graphiques. En collaboration avec mon étudiant Suguru Ishizaki, un système pour la programmation par l'exemple de données dynamiques affichées sur une carte géographique montre la puissance de cet approche pour l'infographie.

Ce travail continue aujourd'hui. Je m'engage dans la réalisation d'un nouveau système pour la programmation par l'exemple, et dans l'exploration de l'application de cette technique à la conception graphique. [Lieberman 89 Fifth-Generation] est un manifeste pour la programmation par l'exemple comme approche à l'augmentation de l'intelligence des outils graphiques. Je présente dans cet article un exemple de la mise en page d'une partition musicale.

La conception graphique

Au MIT, l'équipe Visible Language Workshop a été créée au milieu des années 70 pour étudier l'application de nouvelles technologies de l'informatique à la conception des illustrations graphiques. Le concepteur graphiste Muriel Cooper,

renommée pour ses maquettes de livres et ses mises en page de publications, le photographe Ron MacNeil et plus tard, Patrick Purcell, qui a fait des études sur la conception en architecture au Royal College of Art à Londres, ont animé cette équipe. Ces créateurs étaient les pionniers dans le domaine appelé aujourd'hui la publication assistée par ordinateur. Leur intérêt croissant envers l'application des techniques de l'intelligence artificielle pour aider la conception visuelle, a coïncidé avec mon intérêt d'appliquer mon expertise en intelligence artificielle à ce domaine. En 1987, j'ai rejoint cette équipe, après qu'elle ait été intégrée au Media Laboratory.

A mon avis, la conception graphique devrait susciter un plus grand intérêt chez les chercheurs en intelligence artificielle. Jusqu'à présent, l'intelligence artificielle, et en particulier les systèmes experts, s'appliquait avant tout aux domaines comme les mathématiques, l'ingénierie et la médecine où le raisonnement est principalement de caractère symbolique et verbal. Mais, les techniques de l'intelligence artificielle restent-elles valables dans des domaines où les considérations artistiques et esthétiques sont prioritaires□?

Avec la popularité croissante de stations de travail graphiques, les artistes, les architectes, les animateurs et les concepteurs graphistes disposent maintenant d'outils informatiques. Les possibilités de manipulation d'images deviennent de plus en plus sophistiquées, et l'aide intelligente fournie par l'ordinateur entraînera certainement des bénéfices financiers substantiels. L'intelligence artificielle dispose de nombreuses techniques pour la représentation des connaissances, le raisonnement, et l'apprentissage. Ces techniques ont aujourd'hui atteint leur maturité dans leurs domaines d'étude originels, mais faut-il les modifier pour les appliquer aux domaines graphiques□?

Côté philosophie, de nombreux penseurs sceptiques à l'égard de l'intelligence artificielle croient que celle-ci est forcément limitée aux domaines pour lesquels le raisonnement est précis, bien spécifié, et prévisible. D'après eux, les ordinateurs seraient donc à priori exclus des domaines créatifs et de perception, comme la création artistique et musicale, qui resteraient l'apanage exclusif des humains. A mon avis, cette attitude n'est pas justifiée, mais j'avoue que la question n'a pas encore été suffisamment étudiée. L'étude de l'application de l'intelligence artificielle aux domaines graphiques devrait clarifier les relations entre les méthodologies scientifiques et artistiques.

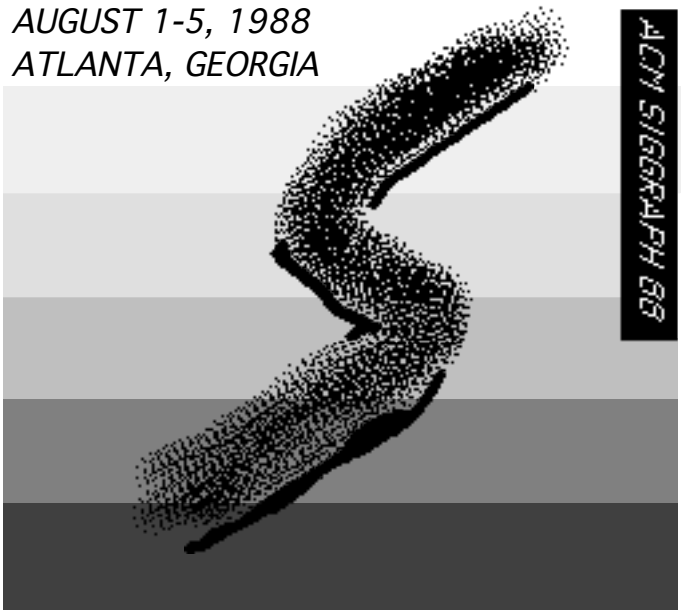
La conception graphique est un domaine qui convient particulièrement à cette étude parce qu'elle est située à mi-chemin entre les deux mondes. Les questions esthétiques y jouent un grand rôle, mais elles ne sont pas les seules à devoir être prises en compte. La conception graphique est toujours au service des objectifs de la communication. Il est souvent possible d'analyser une conception graphique selon des critères de communication relativement objectifs, en tout cas beaucoup moins subjectifs que ceux mis en œuvre pour la peinture, pour la sculpture ou pour la musique. Le vocabulaire graphique utilisé pour la mise en page est relativement restreint, ce qui permet d'étudier les problèmes de résolution de problèmes sans se laisser distraire par le superflu. La conception graphique est un

bon "micro-monde" pour l'étude de la résolution de problèmes dans le domaine graphique.

J'ai fait plusieurs études de génie cognitif [*knowledge engineering*] dans ce domaine, jusqu'à présent pratiquement ignoré par la communauté de l'intelligence artificielle. J'ai essayé d'y analyser le processus de la résolution de problèmes en conception graphique du point de vue informatique. Dans la tradition des systèmes experts, j'ai interviewé des concepteurs graphistes afin de connaître leur style de pensée et leur méthodologie de conception.

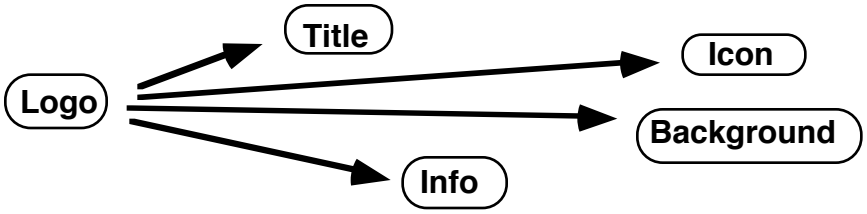
J'ai découvert à la fois des similarités étonnantes et des différences importantes avec les domaines d'application considérés traditionnellement en intelligence artificielle. [Lieberman 88 Fifth-Generation] discute de cette comparaison. Les concepts de l'intelligence artificielle tels que l'arbre des buts, la recherche heuristique, les hiérarchies taxonomiques, ou la résolution de contraintes me semblent valides aussi bien pour les domaines graphiques que pour les domaines symboliques. Mais la connaissance dans les domaines graphiques s'exprime assez difficilement sous la forme de règles SI-ALORS, principal mode d'expression dans les systèmes experts classiques. La résolution de problèmes y est plutôt orientée vers la génération et la critique d'exemples visuels concrets. L'analogie, la généralisation et la résolution de contraintes y jouent un rôle important. Ceci suggère qu'une approche employant la programmation par l'exemple pourrait réussir dans ce domaine.

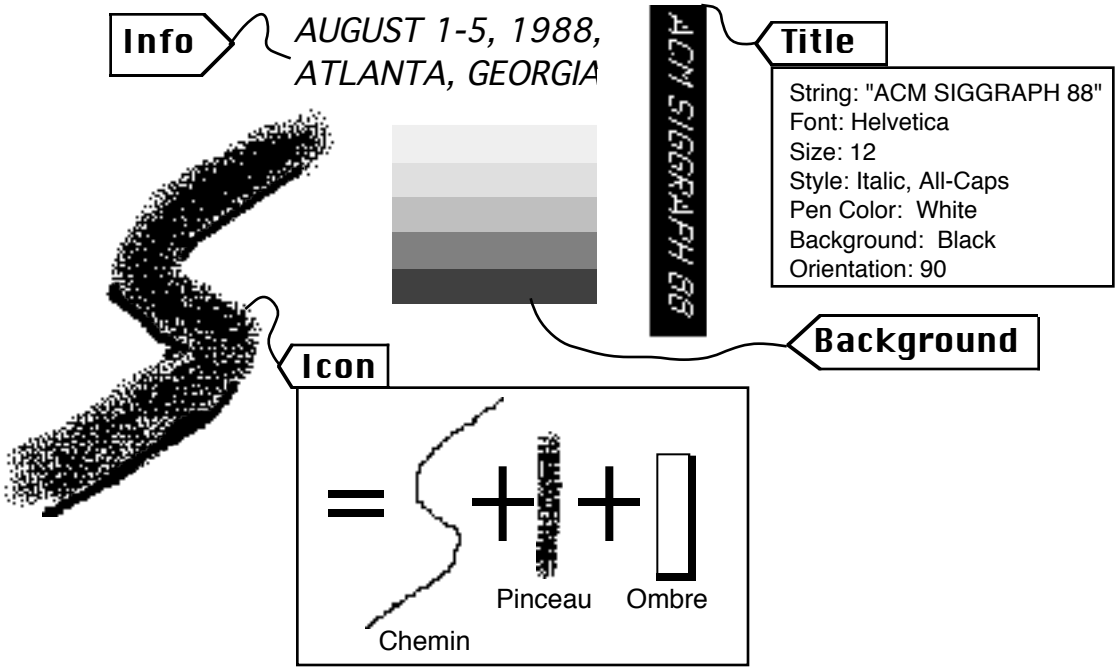
L'acquisition de connaissances en conception graphique est limitée par la difficulté à obtenir les explications verbales provenant des experts. Mais si les experts ont de la difficulté à expliquer leurs raisonnements, par quel moyen est-il possible de transmettre la connaissance entre humains dans ce domaine□? Pour découvrir comment les concepteurs graphistes professionnels transmettent leur savoir, j'ai étudié l'enseignement de la conception graphique aux débutants. Dans [Lieberman 89 Graphic Design] je présente les résultats d'une enquête sur les livres d'introduction à la conception graphique.



En haut, un logotype, et en bas, une analyse de sa représentation

---



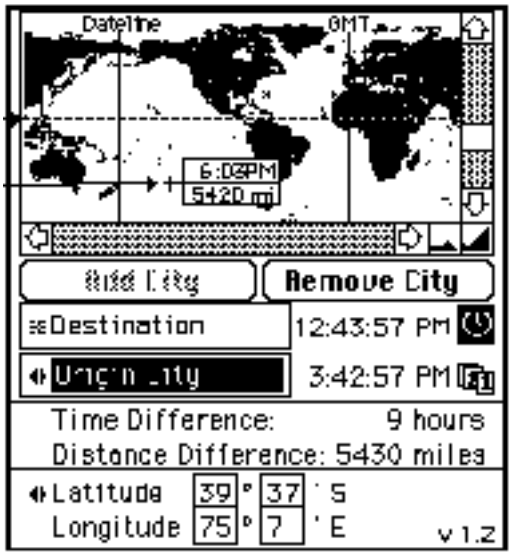


Généralement, les livres décrivent textuellement les principes abstraits et présentent des illustrations de solutions recommandées. J'ai également constaté que presque toute l'information utile se trouve dans les illustrations, plutôt que dans le texte! Mais l'efficacité de l'apprentissage dépend avant tout du processus d'interprétation suivi par l'étudiant. Il faut savoir quels aspects de l'image peuvent être généralisés. Le processus de résolution de problèmes passe souvent par la construction d'une analogie entre le nouveau problème et des études antérieures. Cette analogie est souvent au niveau des buts communicatifs du concepteur. Mon objectif est de créer un système avec lequel un concepteur peut employer un éditeur graphique pour ébaucher une maquette, et expliquer au système pourquoi chaque élément graphique a été choisi. Les principes abstraits pourront être généralisés à partir des exemples visuels concrets. Un tel système devrait être un outil puissant pour l'acquisition de connaissances en conception graphique.

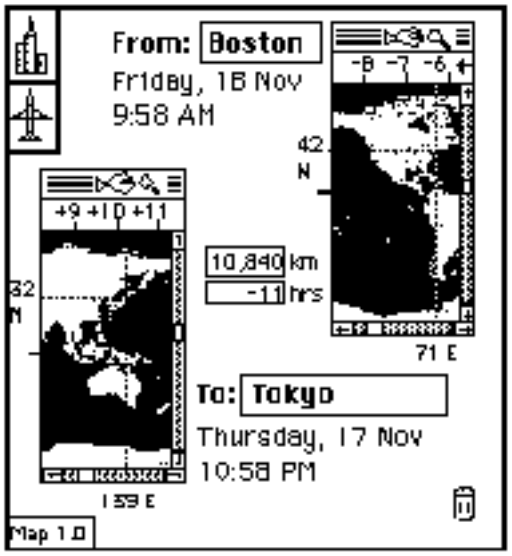
De plus en plus, les concepteurs graphistes sont engagés dans la conception des interfaces graphiques, souvent comme consultants qui critiquent le placement des éléments graphiques sur l'écran. La conception graphique a quitté le milieu statique de la publication sur papier et est entrée dans le milieu dynamique et interactif de l'ordinateur. Souvent, les attitudes divergentes entre les concepteurs graphistes et les informaticiens engendrent des désaccords. Mais personne n'avait comparé sérieusement la méthodologie de conception d'interfaces employée par les programmeurs avec celle employée par les concepteurs graphistes.

En collaboration avec deux concepteurs graphistes d'Apple, nous avons fait une expérience dans laquelle le même problème de conception a été posé simultanément et indépendamment à des concepteurs graphistes et à des programmeurs [Vertelney, Arent, Lieberman 90 Disciplines]. Les concepteurs graphistes ont adopté une stratégie de "conception par critique", orientée vers l'amélioration des fonctions de communication dans l'interface. Ils ont vérifié la conformité de l'interface aux règles de la norme Macintosh [*Macintosh User Interface Guidelines*]. Les programmeurs ont décidé de changer l'interface d'une façon plus radicale, afin de mieux satisfaire aux besoins de l'utilisateur, mais avec l'inconvénient d'éloigner l'interface des règles d'interface Macintosh. Cet article est une contribution à la synthèse entre les deux points de vue, informatique et visuel, qui sera essentielle pour la conception des interfaces à l'avenir.

La version originelle



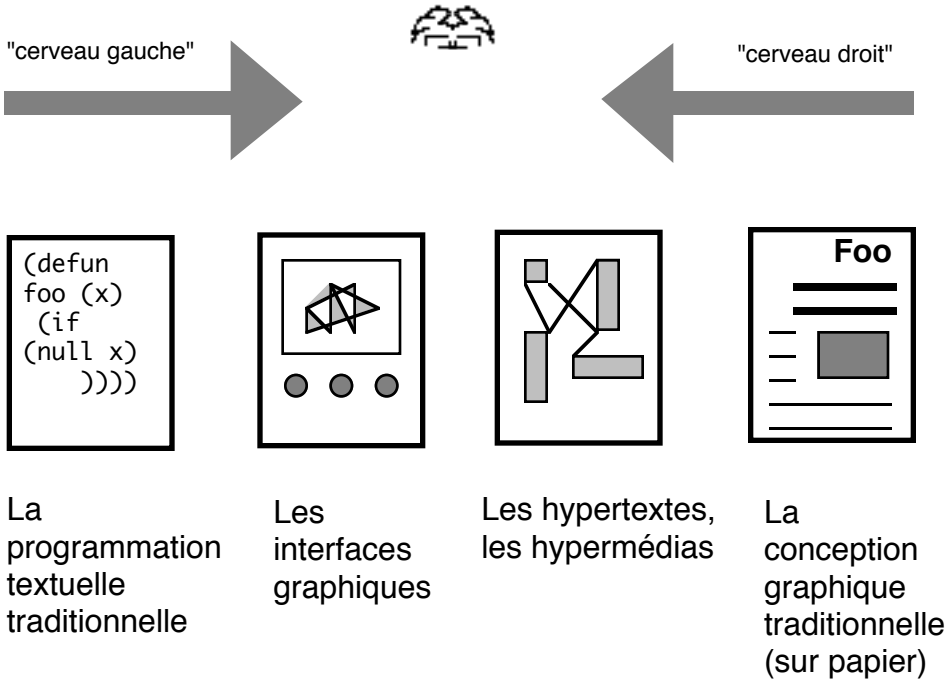
Une version conçue par un concepteur graphiste



Une version conçue par un programmeur

Avec Muriel Cooper et Ron MacNeil, nous avons animé au MIT une série de cours sous la rubrique de "Computer Graphics Workshop". Ce cours interdisciplinaire a essayé d'intégrer des perspectives différentes sur le développement de logiciels graphiques interactifs. Pr. Cooper a collaboré comme concepteur graphiste, et j'ai collaboré comme informaticien. Les lectures pour le cours sont venues des domaines de l'intelligence artificielle, d'interface utilisateur, et de la conception graphique. Les étudiants ont fait des exercises de conception et de programmation d'interfaces graphiques. Nous avons filmé un semestre entier du cours, et j'ai produit un documentaire vidéo [Lieberman, Cooper, MacNeil 89 Poodle]. Pendant mon année sabbatique à l'Université de Paris 6, j'ai enseigné un cours "Intelligence Artificielle et Interface Homme-Machine", qui a partagé les mêmes objectifs que le cours du MIT.

Les *hypermédias* désignent l'information présentée sous la forme d'un réseau d'écrans de texte et de graphique reliés par des liens que l'utilisateur peut suivre interactivement. Les hypermédias font l'objet d'un vif intérêt au Visible Language Workshop actuellement. Les hypermédias sont un point de rencontre entre la conception graphique traditionnelle, les logiciels interactifs graphiques, et l'intelligence artificielle.



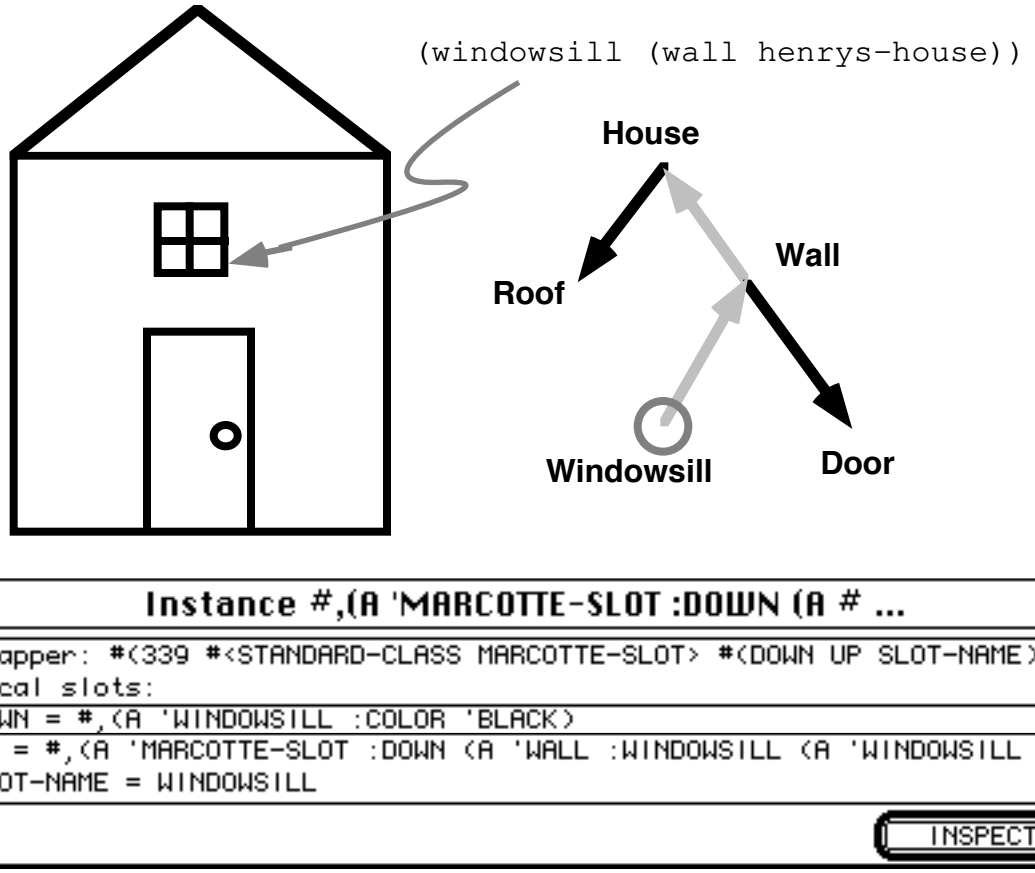
Les concepteurs graphistes traditionnels travaillent sur les livres et les publications, autrefois publiés sur papier, demain affichés sur l'écran de l'ordinateur. Puisque les médias de communication de l'avenir deviendront de plus en plus interactifs, ils approcheront les caractéristiques du logiciel. Puisque les programmes traiteront de plus en plus des réseaux interconnectés d'objets graphiques, ils approcheront les caractéristiques des hypermédias. Un réseau d'objets interconnectés, employé pour les hypermédias, est proche de l'idée de réseau sémantique en intelligence artificielle. Les idées d'inférence et de recherche heuristique dans les réseaux sémantiques seront utiles pour les hypermédias. Des projets sur la synthèse entre les hypermédias, la visualisation de programmes, et les réseaux sémantiques sont en cours avec mes étudiants au MIT.

Le marcottage

Mon séjour au LITP a résuscité mon intérêt pour les techniques de la programmation par objets. J'ai découvert une technique qui trouvera de nombreuses applications dans la programmation par l'exemple, la visualisation de programmes, les hypermédias et la représentation de connaissances. Elle s'appelle le *marcottage*. Cette technique est présentée dans [Lieberman 90 Marcottage]. En botanique, le marcottage est un processus de propagation de plantes, comme des vignes, dans lequel une plante peut s'accrocher dans un autre endroit du sol, et établir de nouvelles racines. On peut considérer le résultat comme une nouvelle plante, dont la "racine" s'est déplacée.

De nombreuses applications telles que les réseaux sémantiques, les réseaux hypertextes, ou les représentations géométriques en infographie, font appel à des réseaux d'objets interconnectés. L'idée de "point de vue" ou de "centre d'intérêt" joue alors un grand rôle. Dans l'approche classique, un réseau est considéré comme quasiment statique, et le centre d'intérêt se déplace au cours du parcours de réseau. Mais il est aussi possible d'envisager une démarche inverse : le point de vue est immobile mais le contexte environnant change, donnant l'impression du déplacement. Cette démarche inédite permet d'implanter plus simplement les algorithmes "navigationnels", pour lesquels la notion de point de vue est plus importante que la position absolue.

Je présente un système dans lequel un objet peut accéder à ses parties constituantes par la technique du *marcottage*. Conceptuellement, chaque fois que l'on suit un lien entre objets, le système "inverse" la direction du lien. Cette technique conduit à un style de programmation "navigationnel", avec lequel le réseau entier reste accessible à partir de n'importe lequel de ses nœuds.



Une représentation marcotte pour une hiérarchie partie-tout d'objets. Une maison est composée d'un toit et d'un mur, le mur ayant une fenêtre et une porte. Une marcotte qui représente la fenêtre est produite par une "inversion virtuelle" des pointeurs. Le butineur en bas affiche un lien UP qui pointe de la fenêtre vers le mur.

Le marcottage facilite la réalisation d'opérations d'annulation ou d'algorithmes réversibles. Une opération de changement incrémental semble être un effet de bord, mais préserve en fait l'état du réseau. Pour maintenir cette illusion, les données sont copiées de façon "paresseuse". Cette technique pourrait être utilisée avec profit pour les annotations personnelles des réseaux hypertextes, le raisonnement hypothétique, et le réaffichage graphique.

Mes projets pour l'avenir

Mon projet à court terme est de continuer le développement de la technique de marcottage. Je collabore avec Pierre Cointe sur une implantation du marcottage en Smalltalk, employant les métaclasses pour modifier le comportement des instances afin d'accéder à leurs variables par marcottage.

J'applique cette technique au problème d'édition de programmes pour la programmation par l'exemple. Dans tous les systèmes précédemment réalisés pour la programmation par l'exemple, le programmeur peut créer des programmes à partir d'exemples, mais doit retourner à la représentation traditionnelle du code pour l'édition. Je voudrais créer un système dans lequel le programmeur peut éditer directement la présentation des exemples pour effectuer une édition du programme. Je représente l'évaluation d'un programme par un parcours par marcottage de sa représentation objet. L'édition peut être effectuée par l'opération de changement incrémental des marcottes.

A moyen terme, je voudrais intégrer la programmation par l'exemple avec la programmation graphique et la représentation de connaissances graphiques. Les générateurs d'interface, comme l'Interface Builder de Jean-Marie Hullot, sont un premier moyen de programmer des interfaces graphiquement. Mais les générateurs d'interface manquent de la puissance procédurale de la programmation conventionnelle. Les techniques de généralisation et d'apprentissage qui se trouvent dans mes systèmes pour la programmation par l'exemple peuvent être appliquées aux générateurs d'interface.

Avec mes étudiants au MIT, nous continuons nos investigations sur la représentation visuelle dynamique de programmes utilisée pour le dépannage. Les études portant sur la visualisation tridimensionnelle de programmes sont en cours, avec la collaboration de mon étudiant David Young. Mes étudiants Michelle Fineblum et Ming Chen explorent la notion traditionnelle de scénario graphique [*storyboard*] de la conception graphique comme représentation dynamique pour les programmes, en intégrant cette idée avec les travaux en hypermédias du Visible Language Workshop.

A long terme, je voudrais continuer ma collaboration avec des concepteurs graphistes, pour étudier la nature de la pensée visuelle et son application aux systèmes informatiques. Je voudrais appliquer mes systèmes pour la programmation par l'exemple comme outil d'acquisition de connaissances en ce domaine. Je voudrais créer des systèmes intelligents pour aider les concepteurs graphistes et les programmeurs à concevoir de nouvelles idées, exprimées graphiquement ou symboliquement, statiquement ou dynamiquement. Je voudrais contribuer à une synergie entre les points de vue visuel et informatique, entre les perspectives scientifiques et artistiques.

Références

Je suis auteur ou co-auteur de toutes ces références suivantes, présentées en ordre chronologique.

[Lieberman, Goldstein, Bochner 74 Logo]
Henry Lieberman, Ira Goldstein, Harry Bochner, "Lisp Logo Manual". AI Memo 206, Logo Memo 11, MIT Artificial Intelligence Laboratory, 1974.

[Lieberman 76 Raster]
Henry Lieberman, "The TV Turtle: A Logo Graphics System for Raster Displays". ACM Symposium on Graphics Languages Miami, Florida, USA, April 1976. Also AI memo 361 and Logo memo 11, MIT Artificial Intelligence Lab.

[Kahn, Lieberman 77 Animation]
Kenneth Kahn and Henry Lieberman, "Computer Animation: Snow White's Dream Machine", Technology Review, October-November 1977.

[Lieberman 78 Coloring]
Henry Lieberman, "How To Color In A Coloring Book". ACM SigGraph Special Interest Group on Computer Graphics Conference, Atlanta, Georgia, USA, August 1978.

[Hewitt, Attardi, Lieberman 79 Guardians]
Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, "Specifying And Proving Properties Of Guardians For Distributed Systems". Conference on Semantics of Concurrent Computing, Evian, France, 1979.

[Hewitt, Attardi, Lieberman 79 Security]
Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, "Security and Modularity in Message Passing". First International Conference on Distributed Computing, Huntsville, Ala., October 1979.

[Lieberman, Hewitt 80 Tinker]
Henry Lieberman and Carl Hewitt, "A Session with Tinker: Interleaving Program Testing With Program Design". Proceedings of the 1980 Lisp Conference, Stanford University. Also AI Memo 577, MIT AI Lab.

[Lieberman 80 Preview]
Henry Lieberman, "A Preview of Act 1". AI Memo 625, MIT Artificial Intelligence Lab, June 1980.

[Lieberman 80 Thinking]
Henry Lieberman, "Thinking About Lots of Things At Once Without Getting Confused". AI Memo 626, MIT Artificial Intelligence Lab, June 1980

[Lieberman 81 Tinker]
Henry Lieberman, "Tinker: Example-Based Programming for Artificial Intelligence". Seventh International Joint Conference on Artificial Intelligence, Vancouver, B. C., Canada, August 1981.

[Lieberman 82 Testing]
Henry Lieberman, "Testing Your Program While You Write It". Workshop on Effectiveness of Program Testing and Proving Methods, Catalina Island, California, USA, May 1982.

[Lieberman 82 Interfaces]
Henry Lieberman, "Constructing Graphical User Interfaces by Example". Graphics Interface '82 Conference, Toronto, Ontario, Canada, May 1982.

[Lieberman 82 Interactive]
Henry Lieberman, "Designing Interactive Systems From the User's Viewpoint". In "Integrated Interactive Computing Systems", P. Degano and E. Sandewall, eds. North Holland 1982.

[Lieberman 82 Music]
Henry Lieberman, "Machine Tongues: Object Oriented Programming". Computer Music Journal, Fall 1982. To appear in a forthcoming book on Object-Oriented Programming for Musical Applications, Stephen Pope, editor, MIT Press, to appear, 1990.

[Lieberman, Hewitt 83 GC]
Henry Lieberman and Carl Hewitt"A Real Time Garbage Collector Based on the Lifetimes of Objects". Communications of the ACM, Vol. 26, No. 6, June 1983. AI Memo 569A, October 1981. Also to appear in a forthcoming book collecting work of the MIT AI Lab Actor group, Carl Hewitt and Gul Agha, eds., MIT Press, 1990.

[Lieberman 83 Apiary]
Henry Lieberman, "An Object-Oriented Simulator for the Apiary". Conference of the American Association for Artificial Intelligence, Washington, D. C., USA, August 1983.

[Lieberman 84 Seeing]

Henry Lieberman, "Seeing What Your Programs Are Doing". International Journal of Man-Machine Studies, Volume 21, No. 4, October 1984. AI Memo 656, MIT AI Lab. Workshop on Intelligent User Interfaces, Jackson, NH, USA, October 1983.

[Lieberman 84 Steps]
Henry Lieberman, "Steps Toward Better Debugging Tools for Lisp". ACM Symposium on Lisp and Functional Programming, Austin, Texas, August 1984.

[Hewitt, Lieberman 84 Parallel]
Carl Hewitt and Henry Lieberman, "Design Issues in Parallel Architectures for Artificial Intelligence". Proceedings of IEEE Compcon Computer Applications Conference, San Francisco, California, USA, March 1984.

[Lieberman 85 Example Video]
Henry Lieberman, "Video Games by Example" [Videotape]. Atari Cambridge Research Center, ACM Siggraph Video Review, No. 12, 1985.

[Lieberman 85 Stepper Video]
Henry Liebeman, "Zstep: A Stepper for Lisp Based on Visual Substitution" [Videotape]. Computers and Human Interaction Conference, 1985. SigGraph Video Review, 1985

[Lieberman 85 Menu]
Henry Lieberman, "There's More to Menu Systems Than Meets the Screen". SigGraph '85 Conference on Computer Graphics, San Francisco, USA, July 1985. AI Memo 831, MIT Artificial Intelligence Laboratory, March 1985.

[Lieberman 86 Encyclopædia]
Henry Lieberman, "Object Oriented Programming Languages". Entry in the Encyclopædia of Artificial Intelligence, Stuart Shapiro, ed. John Wiley & Sons, 1986.

[Lieberman 86 Example]
Henry Lieberman, "An Example Oriented Environment for Beginning Programmers". In Robert Lawler and Masoud Yazdani, eds., Artificial Intelligence and Education., John Wiley & Sons, Chichester, England,1986. AISB Conference on Artificial Intelligence and Education, Exeter, England, September 1985. Instructional Science Journal, Vol. 14, pp. 277-292, 1986.

[Lieberman 86 Act 1]

Henry Lieberman, "Concurrent Object Oriented Programming in Act 1". In "Concurrent Object Oriented Programming", Aki Yonezawa and Mario Tokoro, eds., MIT Press, 1986.

[Lieberman 86 Delegation]
Henry Lieberman, "Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object Oriented Systems". 3èmes Journées d'études sur les langages à objets, P. Cointe et J. Bézivin, eds., AFCET, Paris, France, January 1986.

[Lieberman 86 Interfaces Video]
Henry Lieberman, "Ezwin: a Kit for Building Menu-Based Graphical Interfaces" (video tape). SigCHI [Computers & Human Interaction] Conference, Boston, April 1986. SigGraph Video Review, 1986.

[Lieberman 86 Prototypes]
Henry Lieberman "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". First Conference on Object-Oriented Programming Languages, Systems, and Applications [OOPSLA-86], ACM SigCHI, Portland, Oregon, USA, September 1986.

[Lieberman 86 Debugging]
Henry Lieberman, "Debugging Graphical User Interfaces". ACM SigGraph Workshop on Software Tools for User Interfaces, Seattle, Washington, November 1986.

[Lieberman 87 Reversible]
Henry Lieberman, "Reversible Object-Oriented Interpreters". European Conference on Object-Oriented Programming, Paris, France, P. Cointe, J. Bézivin, J. M. Hullot, H. Lieberman, eds., Springer-Verlag, 1987.

[Lieberman 87 Unpredictable]
Henry Lieberman, "Expecting the Unpredictable: When Computers Can Think in Parallel". In "Matters of Intelligence", Lucia Vaina, ed., D. Reidel, 1987.

[Lieberman 87 Synthesizer]
Henry Lieberman, Casio CZ-1 Synthesizer: A User's Report. Computer Music Journal, Summer 1987.

[Lieberman 88 Parallelism]
Henry Lieberman, "Parallelism in Interpreters for Knowledge Representation Languages". In Concepts and Characteristics

of Knowledge Based Systems, Mario Tokoro, ed., North-Holland 1988.

[Lieberman, Fry 88 Common EVAL]
Henry Lieberman and Christopher Fry, "Common EVAL". Lisp Pointers, July-September, 1988.

[Lieberman 88 Fifth-Generation]
Henry Lieberman, "Towards Intelligent Interfaces for Graphic Design Applications". International Conference on Fifth Generation Computer Systems, Tokyo, Japan, November 1988.

[Stein, Lieberman, Ungar 89 Treaty]
Lynn Stein, Henry Lieberman and David Ungar, "The Treaty of Orlando: A Shared View of Sharing". In Object-Oriented Concepts, Applications and Databases, Won Kim and Fred Lochovsky, eds., Addison-Wesley, 1989.

[Lieberman 89 Analogies]
Henry Lieberman, "Programming Descriptive Analogies by Example". Proceedings of the Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages, Viareggio, Italy, Maria Simi and Daniele Nardi, eds., February 1989.

[Lieberman 89 CHI]
Henry Lieberman, "User Interfaces for Program Debugging". Tutorial at ACM Computers and Human Interaction Conference [CHI '89], Austin, Texas, USA, ACM Press, April 1989.

[Lieberman 89 Graphic Design]
Henry Lieberman, "Communication of Expert Knowledge in Graphic Design" , Visible Language Workshop Research Report, MIT Media Laboratory, 1989. To be submitted to Conference on AI in Design, Edinburgh, Scotland, 1991.

[Lieberman, Cooper, MacNeil 89 Poodle]
Henry Lieberman, Muriel Cooper and Ron MacNeil, "Is it a Poodle?", Videotape, 3 minutes, color. Shown at Media Lab presentation at SigGraph 89, Boston.

[Lieberman 89 3D]
Henry Lieberman, "A Three-Dimensional Representation for Program Execution". IEEE Conference on Visual Languages, Rome, Italy, October 1989.

[Lieberman 89 Visual Programming]

"Visual Programming: A Vision for the Future". Friend-21 Conference on Next Generation Human Interface Technologies, Tokyo, Japan, September 1989.

[Vertelney, Arent, Lieberman 90 Disciplines]
Laurie Vertelney, Michael Arent and Henry Lieberman, "Two Disciplines in Search of an Interface". In The Art of Human-Computer Interface Design, Brenda Laurel, ed., Addison-Wesley, August 1990.

[Lieberman 90 Marcottage]
Henry Lieberman, "Le Marcottage: Une Approche Navigationnelle de la Programmation Par Objets". Rapport de Recherche 90-?, LITP/RXF, Université de Paris VI, septembre 1990. [Une version anglaise sera envoyée à ECOOP-91].