HAL
open science

Notation et représentation formelle des processus de composition

Dominique Fober, Stéphane Letz, Yann Orlarey

To cite this version:

Dominique Fober, Stéphane Letz, Yann Orlarey. Notation et représentation formelle des processus de composition. [Rapport Technique] GRAME. 1992. hal-02158785

HAL Id: hal-02158785

https://hal.science/hal-02158785v1

Submitted on 18 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Notation et Représentation Formelle des Processus de Composition

Rapport de Recherche

auprès du Département de la Recherche de la Direction de la Musique

du Ministère de l'Education Nationale et de la Culture

∞

D. Fober - S. Letz - Y. Orlarey / Grame / Octobre 92

GRAME, 6 QUAI JEAN MOULIN, 69001 LYON, FRANCE

SOMMAIRE

1. INTRODUCTION 1

2. CARACTERISATION DES PARADIGMES DE PROGRAMMATION 4
   2.1. Eléments de caractérisation 4
   2.2. Analyse de différents paradigmes de programmation 6
   2.3. La représentation du temps 15

3. LE PRINCIPE D'HOMOGENEITE 16
   3.1. Définition du principe d'homogénéité 16
   3.2. Un exemple de mise en oeuvre du principe d'homogénéité : le G-CALCUL 17

4. VERS UN ESPACE DE COMPOSITION HOMOGENE 28
   4.1. Les bases d'un langage de composition homogène : le T-Calcul 28
   4.2. Définition d'un espace de composition musicale. 33
   4.3. Définition d'un nouveau type de document musical. 35
   4.4. Architecture générale 40

5. CONCLUSION 42

6. ANNEXES 44
   6.1. Le Lambda Calcul 44
   6.2. Implémentation Common Lisp du G-Calcul 49

7. BIBLIOGRAPHIE 65

1. INTRODUCTION

L'évolution de la notation musicale, du IXe siècle à nos jours, est indissociable des transformations et des mutations des pratiques musicales. Elle en est à la fois le reflet et l'un des moteurs essentiels. La possibilité de noter, par des signes conventionnels et abstraits, les phénomènes musicaux et leurs relations temporelles a favorisé une séparation progressive et une professionnalisation des métiers de compositeurs et d'interprètes. Elle a bouleversé l'enseignement et l'apprentissage de la musique. Elle a permis la constitution et la transmission d'un immense patrimoine culturel, toujours vivant, car toujours étudié, interprété, enrichi et re-créé. De plus, et c'est ici un point qui nous intéresse tout particulièrement, la notation musicale s'est avérée être un outil cognitif essentiel. En prolongeant et en amplifiant les capacités mentales de mémorisation et de représentation du compositeur, la notation musicale a contribué à l'élaboration d'une pensée dont la sophistication et la complexité n'ont cessé de croître.

Pourtant, la lente maturation, qui au cours des siècles a constitué la notation musicale en un modèle sophistiqué et efficace de représentation, est aujourd'hui bousculée par l'évolution des supports et des pratiques contemporaines. Le développement des musiques électro-acoustiques a montré les faiblesses de la notation traditionnelle pour appréhender et décrire les phénomènes sonores avec précision. Cet état de fait a conduit de nombreux compositeurs à développer des systèmes spécifiques de notation, parfois propres à une seule oeuvre, sans que se dégage aucune forme de normalisation. L'apparition des synthétiseurs a encore compliqué cette situation, en faisant passer les paramètres sonore du domaine du sensible à celui d'une méthode de synthèse arbitraire et tributaire de l'évolution technologique. D'une manière générale, la dépendance de plus en plus étroite de nombreuses oeuvres contemporaines vis à vis de moyens technologiques condamnés à disparaître à plus ou moins brève échéance est inquiétante. Il est peu probable que puisse se constituer un quelconque patrimoine culturel sur des bases aussi mouvantes.

Dans l'état actuel, le recours aux langages informatiques de composition musicale ne résout pas le problème. Bien que d'une grande généralité, ces outils restent marginaux et peu diffusés. Ils sont d'un abord difficile car en rupture avec les connaissances habituelles du compositeur. Celui-ci se trouve confronté à deux "espaces" de travail très différents. D'une part la "partition électronique", assez rapidement familière, qui permet de visualiser et d'éditer les matériaux musicaux de la composition. D'autre part l'environnement de programmation, plus difficile à appréhender, qui permet de décrire les processus compositionnels, mais également de générer, de manipuler les objets de la composition.

Ainsi, alors que notation musicale et processus compositionnels participent d'une même essence, les langages informatiques de composition musicale introduisent souvent un triple clivage : les espaces de travail sont séparés, les modalités de structuration et d'abstraction sont différentes, les modes de représentation ne sont pas du tout les mêmes. De plus, ils induisent un glissement de sens entre la partition et le programme informatique, ce dernier, bien qu'inadapté, devenant la "véritable" partition, celle par laquelle le compositeur conçoit et comprend son oeuvre. Il faut ajouter à cela l'hétérogénéité des approches et des notations adoptées par les langages informatiques de composition musicale ainsi que leur forte dépendance à divers supports matériels et logiciels.

Nous constatons donc que les avancées technologiques de cette fin de XX° siècle, tout en élargissant le champ des possibles ont créé parallèlement de nouveaux besoins d'écriture et de représentation qui ne peuvent être pris en compte par la notation musicale traditionnelle. En l'absence d'une structuration communément admise, capable d'abstraire l'oeuvre musicale de ces dépendances technologiques, il devient difficile de constituer un patrimoine culturel vivant, se prêtant à l'étude et à la transmission d'un savoir, susceptible d'être repris à son compte par les générations futures.

Peut-on résoudre ce problème par une notation musicale élargie à nos besoins actuels ? Nous ne ferions probablement là que reporter le problème de quelques années, compte tenu de la phase de mutations technologiques accélérées dans laquelle nous somme désormais entrés. Par conséquent, il ne s'agit pas tant de définir une notation musicale figée, même élargie, mais bien de définir des modalités d'évolution de cette notation, sans en connaître à priori les évolutions possibles. Il est intéressant de noter que cette question n'est pas limitée au seul domaine musical. Elle concerne l'ensemble des "documents électroniques" dont on souhaiterait qu'ils restent pleinement exploitables dans le futur alors même que les logiciels et machines qui auront permis leur constitution auront disparu depuis fort longtemps.

Dans ce contexte, et si l'on se restreint au domaine de l'informatique, il n'existe pas d'impossibilité théorique à définir un système de notation continûment extensible, dont on ne puisse garantir qu'il permettra de prendre en compte tous les besoins futurs. On connait depuis les années trente, des systèmes formels tels que les "Machines de Turing" ou le Lambda-Calcul qui permettent d'exprimer toute fonction pouvant être calculée par un procédé mécanique (et donc par un ordinateur). Notre problème est alors avant tout d'ordre pratique : un formalisme, un langage de programmation sont porteurs d'une "ergonomie" permettant d'appréhender et de résoudre certains types de problèmes plutôt que d'autres. Il convient donc, dans la définition d'un langage d'assistance à la composition musicale, de cerner au

mieux sa future "ergonomie", afin de garantir non seulement son adéquation à la formalisation de problèmes musicaux, mais également son accessibilité à une pensée qui soit musicale et non pas informatique.

Nous nous proposons de nous interroger sur les conditions et les répercussions d'un mariage réussi entre notation musicale et langages informatiques. Par mariage réussi, nous entendons une notation musicale étendue à la description et à la représentation de processus compositionnels, tout en s'inscrivant dans la continuité de la notation musicale traditionnelle. Cette question constitue le thème central du présent rapport qui, sans prétendre à l'exhaustivité, apporte différents éléments de réponse.

On trouvera tout d'abord une étude des principaux paradigmes de programmation actuels dans le but de comprendre en quoi un langage de programmation est porteur d'une certaine "ergonomie cognitive" qui le rend plus ou moins adapté à certains domaines d'application.

Cette étude nous permettra de formuler un principe d'homogénéité qu'il nous parait important de prendre en compte dans la définition d'un langage de programmation. Nous en présenterons une mise en oeuvre dans le domaine graphique.

Enfin nous esquisserons les grandes lignes d'un espace de composition musicale basé sur le principe précédent en nous préoccupant également des problèmes de normalisation et de pérennité des documents musicaux.

∴

2. CARACTERISATION DES PARADIGMES DE PROGRAMMATION

De nombreux travaux ont été réalisés dans le domaine des langages informatiques pour la synthèse sonore et la composition musicale. Parmi les plus connus on peut citer : MUSIC I à V [Mathews 61], Formes [Cointe, Rodet 83], PLA [Schottstaedt 83] et MAX [Puckette 88]. Notre équipe a également contribué à ce domaine avec : MidiLogo [Orlarey 84], MidiLisp [Boynton et al. 86] et CLCE [Letz et al. 90].

Tous ces travaux témoignent d'une grande diversité d'approches. Néanmoins, au delà de leurs différences, ils visent un objectif commun : mettre au service du compositeur la dimension de machine universelle, programmable, de l'ordinateur. Pourtant, comme nous l'avons souligné, les langages informatiques de composition musicale restent encore d'un abord difficile car trop souvent en rupture avec les modes de pensées et les connaissances habituelles du compositeur.

En effet, un langage informatique, malgré son "universalité" est porteur d'une "ergonomie cognitive" qui influence la manière dont nous pouvons penser ou conceptualiser un problème donné [Dijkstra 72]. Il convient donc de chercher à comprendre en quoi tel ou tel langage est intrinsèquement "meilleur" dans son adéquation avec certains types de pensées et certains domaines d'application. Les caractéristiques plus techniques (efficacité, vitesse...) ne seront donc pas prises en compte ici.

2.1. ELEMENTS DE CARACTERISATION

A priori, la plupart des langages informatiques sont équivalents dans le sens où ils peuvent calculer toute fonction calculable. Cependant, les facilités d'expression, de représentation et d'utilisation diffèrent largement de l'un à l'autre en fonction du domaine d'application. Ce qu'il est alors important de dégager, ce sont les éléments qui relient le domaine formel du langage au domaine sémantique tel que le conçoit l'utilisateur. Le problème peut être formulé de la manière suivante :

- En quoi les langages informatiques permettent-ils d'exprimer des problèmes complexes ?

- Quels sont les éléments qui caractérisent la puissance d'un langage ?

- En quoi un langage est-il adapté à son domaine d'application ?

- Dans quel mesure cette adaptation, plus ou moins bonne, modifie-t-elle la manière de penser et de conceptualiser de son utilisateur ?

Pour répondre à ces questions, nous proposons de définir deux concepts, communs à la plupart des langages de programmation: l'*abstraction* et la *construction*, ainsi que la notion de *distance téléologique*. Ces éléments nous serviront par la suite de grille d'analyse pour différents paradigmes de programmation actuels.

2.1.1. Abstraction

La notion d'abstraction représente la possibilité de décrire et de manipuler une entité sans avoir besoin de préciser tous ses constituants ou toutes ses propriétés internes. C'est un mécanisme qui permet d'introduire une certaine généralité dans une entité, en la *détachant* d'un ou plusieurs éléments qui lui servent de support. Un entité ainsi abstraite pourra ensuite être interprétée suivant le cas comme une relation, une fonction, une classe, un ensemble etc. Un exemple courant d'abstraction dans les langages de programmation est la possibilité de définir de nouvelles fonctions en partant d'expressions dont on rend "variables" certaines parties. L'abstraction peut également concerner la description des données, voir des types de données, manipulés par les programmes.

2.1.2. Construction

Lorsqu'on aborde un problème difficile, on cherche souvent à le diviser en sous-problèmes plus simples, à résoudre les sous-problèmes et à combiner les solutions. Pouvoir décomposer un problème complexe en parties plus simples dépend directement de la manière dont on pourra ensuite assembler les solutions de chaque partie. Les capacités de construction d'un langage informatique sont donc essentielles et en étroite relation avec ses facilités d'abstraction. Ensemble, ces deux notions conditionnent en grande partie la souplesse, la généralité et la modularité d'un langage. Suivant les paradigmes de programmations on trouvera différentes méthodes de construction tels que la séquence et la répétition d'instructions, la combinaison de structures de données, l'application et la composition de fonctions ou encore la connexion de modules.

2.1.3. Distance téléologique

Cette notion abstraite, liée à la perception humaine, cherche à donner une mesure de l'*effort mental* que doit produire le programmeur pour passer de

l'*idée* d'un programme à sa description explicite dans un langage donné. Un langage de programmation sera d'autant mieux adapté à résoudre certains types de problèmes qu'il facilitera le passage de la spécification du problème au code qui lui correspond. On peut également considérer la distance inverse, l'effort mental nécessaire pour comprendre ce que représente le code d'un programme. Nous dirons dans ce cas qu'un langage de programmation *est un système de notation et de représentation* d'autant plus efficace qu'il minimise cette distance inverse.

2.2. ANALYSE DE DIFFERENTS PARADIGMES DE PROGRAMMATION

L'histoire de l'informatique est généralement envisagée sous l'angle du progrès matériel et électronique des ordinateurs, regroupés en génération successives. Dans cette optique, les langages de programmations sont essentiellement présentés comme moyen de contrôle de la machine. Nous adoptons ici le point de vue dual. Nous envisageons les langages de programmation comme des outils de pensée et de conceptualisation, et les machines physiques comme supports de ces langages. L'évolution des langages de programmation est alors révélatrice de paradigmes sous-jacents quant à la *bonne façon* de construire des modèles et de résoudre des problèmes.

2.2.1. Repères historiques

Les fondements théoriques de l'informatique ont été définis dans les années 30 avant même l'invention du premier ordinateur. En 1936, les travaux d'Alan Turing sur les nombres calculables et le problème de la décidabilité de Hilbert l'amène à formaliser très précisément la notion de méthode effective de calcul, à en montrer les limites et à proposer une machine conceptuelle *universelle*, capable de calculer tout ce qui est mécaniquement calculable. A la même époque, Alonzo Church travaille également sur le problème de la décidabilité et propose un formalisme très élégant, le Lambda-Calcul qui permet, comme la machine de Turing, de calculer toute fonction calculable. Ces deux formalismes, machine de Turing et Lambda-Calcul, sont à la base de deux paradigmes importants : la programmation impérative et la programmation fonctionnelle.

Il fallut attendre le milieu des années 40 pour voir apparaître les premiers ordinateurs opérationnels. La programmation s'effectuait alors dans le "langage" même de la machine. L'une des première idées marquante fut de considérer que le programme devait être à l'intérieur de la machine, au même titre que les données. Cette approche permettait d'envisager des programme capables de se modifier, voir même *d'apprendre*. Une autre innovation

importante fut l'invention de la notion de sous-routine qui permettait de représenter un programme par un code, qui pouvait à son tour être utilisé dans un autre programme et ainsi de suite, réduisant considérablement la complexité de l'ensemble.

En 56, un pas important est franchis avec FORTRAN (Backus), premier langage évolué. FORTRAN permet de penser les problèmes directement en termes de formules et de variables et non plus en code machine. Il réduit considérablement la distance téléologique pour tous les problèmes de calculs mathématiques, par rapport au langage machine. En 1959 apparut LISP (McCarthy), un langage qui reste encore aujourd'hui d'une importance conceptuelle primordiale. Basé sur le paradigme fonctionnel, LISP propose la notion de liste comme structure de données générale, utilisée à la fois pour représenter les données et les programmes.

Un an plus tard, la publication de la charte ALGOL 60 marque la volonté de la communauté internationale de définir un langage impératif standard et universel. Peu de temps après apparait le premier langage de synthèse musicale MUSIC (Mathews), qui est également l'un des tout premier langages flot de données !

Les années 70 sont riches en événements marquants avec le langage C (Ritchie), mais également avec l'apparition de nouveaux paradigmes de programmation : programmation en logique avec PROLOG (Colmerauer), programmation objet avec SMALLTALK (Kay), la programmation par acteurs (Hewitt). En 1978, un article de Backus, [Backus 78] le père de FORTRAN, relança l'attention des chercheurs sur l'intérêt de l'approche fonctionnelle et de manière plus générale sur la nécessité de réfléchir à de nouveaux paradigmes de programmation.

Les années 80 voient se développer considérablement la problématique du parallélisme tant au niveau de l'architecture des machines que des modèles de programmation CSP (Hoare), CCS (Milner). La paradigme de la programmation objet se renforce et passe dans les moeurs grâce notamment à C++ (Stroustrup). Des efforts sont réalisés dans le domaine de la normalisation des langages avec ADA, la norme ANSI de C et la norme ANSI de Common Lisp. D'anciens paradigmes de programmation sont revisités et suscitent quantités de travaux, tels que les Automates cellulaires, les réseaux neuro-mimétiques ou encore les algorithmes génétiques.

A la lumière de ce rapide historique, il apparait un enrichissement et une diversification des paradigmes de programmation utilisés. Ceux-ci ne sont pas toujours exclusifs les uns des autres et certains langages peuvent se réclamer de plusieurs paradigmes à la fois. Comme nous l'avons dit, chaque

paradigme de programmation est porteur d'une ergonomie cognitive qui conditionne très fortement les modes de conceptualisation qui lui sont rattachés et par là même les classes de problèmes qui lui sont "naturels". Nous allons donc essentiellement réfléchir sur la signification et sur les facilités de modélisation de cinq grands paradigmes : impératif, objet, flots de données, logique et fonctionnel.

2.2.2. les langages impératifs

Ce sont les premiers langages, écrits dans le but direct de contrôler le fonctionnement des ordinateurs. Dans un domaine général, on peut citer

Fortran, Pascal, C, ADA, etc. et dans le domaine musical MidiLogo ou MidiLisp. Ils sont caractérisés par :

- la présence d'un état implicite manipulé par les commandes du langage.
- une notion de séquence d'instructions permettant le contrôle précis de cet état.
- une opération d'affectation utilisée pour changer l'état de la machine.

Les instructions sont enchaînées sous forme de séquences à l'aide de structure de bloc (begin ... end), les transferts de contrôle sont réalisés à l'aide de commandes explicites (goto, boucles), l'opération d'affectation est utilisée pour changer l'état de la machine. Ainsi ces langages sont caractérisés par un ensemble de structures fixées que le programmeur doit utiliser pour définir ses propres procédures.

Capacité d'abstraction

Elle est réalisée par la définition de fonctions ou de structures de données abstraites mais est en partie limitée par une grande différence entre les programmes et les données, qui ne permet pas la manipulation d'objets fonctionnels.

Constructibilité

La construction de gros systèmes se réalise essentiellement à l'aide du mécanisme de définition de fonction ou de types de données abstraites.

Distance téléologique

Il est évidemment difficile de donner des réponses tranchées dans ce domaine. On peut remarquer cependant que de manière générale, les langages impératifs sont plutôt adaptés à la description de processus par nature séquentiels ou à la modélisation de machines à états. Dans ce cadre précis il peut alors exister une bonne adéquation entre le programme et ce qu'il représente. Mais il existe un ensemble de problèmes pour lesquels l'existence d'un état et d'effets de bords, rend la manipulation et le raisonnement sur les programmes plus difficiles et augmente la distance téléologique. La distance téléologique inverse peut être très grande et nécessiter de simuler mentalement le fonctionnement du programme pour tracer l'évolution des états internes.

Domaines d'application

Les langages impératifs sont donc, à cause de leur structure intrinsèque, plutôt adaptés à la description de processus séquentiels ou à la modélisation de machines à états. Leur "pouvoir" de représentation dans d'autres domaines est limité par une structure opérationnelle figée, par la difficulté à pouvoir manipuler formellement le langage où le sens d'un programme n'est pas toujours explicite dans sa forme textuelle.

2.2.3. Les langages orientés objets

On peut citer dans un domaine général, Smalltalk, C++, CLOS, et Formes, CLCE [Grame 1990] dans le domaine musical.

La programmation orientée objet est basée sur l'idée de transposition des objets du monde réel sous forme d'objets informatiques. Les connaissances déclaratives (données) et procédurales (programmes) qui modélisent un objet sont regroupées dans une même entité. Ils permettent des abstractions de haut niveau par l'intermédiaire des notions d'héritage et d'instanciation. Les problèmes du monde réel sont modélisés sous la forme d'un graphe de classes d'objets dont les comportements vont du plus général au plus spécialisé : une classe fille *hérite* des caractéristiques de sa classe mère. Chaque classe est un moule à objets et chaque objet possède les comportements de sa classe et des classes ascendantes, mais possède un état interne qui lui est propre. Le mécanisme opérationnel est souvent assuré par un système de messages où les objets reçoivent et émettent des commandes à destination d'autres objets. Enfin, il existe une notion de polymorphisme, représentant la capacité d'une même opération à prendre plusieurs formes.

Capacité d'abstraction

l'idée d'abstraction est présente à 3 niveaux :

- dans l'objet qui peut être considéré comme une boite noire réalisant une abstraction sur des données et des comportements,

- dans la notion d'héritage où les comportements les plus généraux sont abstraits dans les classes de niveau supérieur,

- par le mécanisme de polymorphisme où l'on peut par exemple, définir une opération qui aura une "interprétation" différente suivant les objets qui l'exécuteront.

Constructibilité

L'activité de construction d'un programme se transpose directement dans la construction du graphe des classes qui modélisent le système. Cette constructibilité est directement liée aux capacités d'abstraction du langage.

Distance téléologique

Le programme est une représentation directe du système réel qu'il modélise. La distance téléologique est donc réduite puisque le passage mental de la représentation informatique au monde réel est plus simple, ceci évidemment si le système à représenter se prête à une interprétation directe.

Domaines d'application

De manière générale, la programmation orientée objet est une avancée tant dans le sens des possibilités d'abstraction et de construction, que de la réduction de la distance téléologique. Ce type de langage, adapté à la représentation de problèmes concrets sous forme d'ensembles de machines à état en communication, se prête peu en revanche à la vérification formelle des programmes (par exemple l'égalité de fonctionnement de deux programmes différents).

2.2.4. Les langages flot de données

On peut citer dans un domaine temps-réel : LUSTRE [Halbwachs 91] et ESTEREL [Berry 87] , dans le domaine musical : MUSIC V [Mathews 61] et MAX.

Dans les modèles impératifs, les opérations sont décrites en termes de séquences d'instructions qui manipulent des données contenues dans des registres ou des positions mémoire. Les architectures de ce type ont donc un modèle d'évaluation défini en termes de "flot de contrôle".

L'autre possibilité est de définir un modèle où ce sont les données qui sont les entités actives. Elles circulent parmi une collection d'opérateurs passifs qui effectuent les manipulations lorsque toutes les données nécessaires sont disponibles. Ainsi un programme peut être vu comme un réseau d'opérateurs agissant en parallèle au rythme de leurs entrées. Il est surtout utilisé dans les systèmes réactifs, dans le domaine industriel, où sa "philosophie" est en accord avec les méthodologies classiques utilisées en automatisme.

Dans le cas où les noeuds du réseau sont des objets fonctionnels (sans états), le modèle complet est fonctionnel, et le formalisme possède des caractéristiques mathématiques qui permettent l'application de méthodes formelles pour construire, transformer et vérifier les programmes.

Capacité d'abstraction

Construire un réseau est l'équivalent de la définition de fonctions dans un langage classique, l'abstraction est réalisée lorsqu'on manipule un réseau comme un tout en le considérant comme une boite noire avec des entrées et des sorties. De plus, il n'y a pas d'impossibilité théorique à la réalisation d'abstractions de plus haut niveaux sur des fonctions.

Constructibilité

L'opération de construction consiste à assembler des sous réseaux considérés comme des opérateurs. Elle est conceptuellement identique à la construction d'opérations complexes dans les langages textuels lorsqu'on définit des fonctions qui utilisent comme argument le résultat d'autres fonctions et dont le résultat peut être utilisé comme argument par la suite.

Distance téléologique

La description sous forme de réseaux d'opérateurs fournit directement une représentation graphique des programmes [Halbwachs 91]. De plus cette représentation présente très naturellement une notion de décomposition hiérarchique : un sous-réseau peut être considéré comme un opérateur. Comme il y a équivalence entre le formalisme textuel et le formalisme graphique, il est possible de combiner les avantages des deux modes de représentation.

Domaines d'application

Le modèle flot de données est par essence adapté à la description de systèmes réactifs temps réel. Par sa structure mathématique "propre", il se prête à des manipulations formelles. De plus l'équivalence entre le formalisme textuel et le formalisme graphique est une caractéristique allant dans le sens d'une réduction de la distance téléologique.

2.2.5. Langages logiques

On peut citer dans un domaine général les multiples versions de PROLOG et de ses dérivés, et dans le domaine musical les expérimentations de j. Vandenheede à l'IRCAM.

Ils sont basés sur un modèle de programmation déclarative dont le modèle sous-jacent de calcul est la relation. On distingue trois niveaux en matière de traitement d'expressions :

- la spécification de faits et de règles, c'est à dire de connaissances permanentes relatives au domaine concerné,
- la spécification du problème à résoudre,
- un mécanisme de résolution opérant sur les deux premiers niveaux.

Le programmeur ne spécifie plus la procédure de résolution à employer de manière explicite : c'est le système sous-jacent qui se charge de résoudre le problème en utilisant les informations dont il dispose. L'activité de programmation consiste donc à enrichir le système d'informations définissant un domaine, pour pouvoir ensuite résoudre les problèmes qui en relèvent.

Capacité d'abstraction

Dans un langage logique, au contraire des langages classiques où elles désignent une valeur connue, les variables représentent des arbres inconnus. Le déroulement d'un programme ne vise pas à instancier les variables pour produire un résultat mais à les déterminer de façon à vérifier les règles dont elles font parties. Ainsi l'opération d'abstraction se réalise dans la définition de règles comprenant des parties variables qui expriment une connaissance générale sur un ensemble d'objets.

Constructibilité

Puisque les langages logiques manipulent des faits logiques c'est à dire ayant une valeur de vérité, l'opération de construction est basée sur l'utilisation d'opérateurs logiques : et , ou , non, relation d'implication ou d'équivalence. Elle est encore limitée par la difficulté à gèrer des "mondes" de connaissances différents et à restreindre la portée des règles.

Distance téléologique

Les langages logiques sont basés sur un système formel qui modèlise un certain nombre de processus du raisonnement humain. Pour toute une classe de problèmes dont le mécanisme opérationel relève de l'utilisation de règles de déduction logique, le programmeur n'a plus besoin de spécifier explicitement le processus de résolution, il lui suffit de fournir au système un certain nombre de connaissances de manière déclarative. Ainsi la distance téléologique est réduite puisqu'on passe plus facilement de la spécification du problème à son expression dans le langage en supprimant l'écriture explicite des mécanismes de contrôle.

Domaines d'application

Les langages logiques sont particulièrement bien adaptés à l'ensemble des problèmes relevant du traitement ou de la manipulation des connaissances: bases de données, systèmes formels, démonstrateurs de théorèmes, systèmes de réécriture..etc.. Ils présentent néanmoins certaines insuffisances quant aux possibilités d'abstraction et de modularité. De plus, à cause de la spécificité des problèmes qu'ils permettent de résoudre, ils sont difficilement utilisables dans d'autres domaines d'application.

2.2.6. Les langages fonctionnels

On peut citer dans un domaine général : HOPE, ML et Haskell, dans le domaine musical : Artic.

Les langages fonctionnels sont des modèles de programmation par expression dont le modèle de calcul sous-jacent est la fonction. La programmation par expressions présente les caractéristiques suivantes :

- indépendance de l'ordre d'évaluation : l'ordre d'évaluation des sous-expressions d'une expression n'est pas déterministe, ce qui facilite l'implémentation sur des architectures parallèles.

- transparence référentielle : les sous-expressions ont la propriété de pouvoir être remplacées par leurs valeurs indépendamment de l'expression externe.

- sens des expressions explicite dans leur forme textuelle : les arguments des opérateurs et leurs résultats sont complètement décrits dans leur forme textuelle.

- absence d'effets de bords : les expressions sont indépendantes de tout contexte.

Ces caractéristiques des expressions fonctionnelles permettent d'effectuer des raisonnements formels sur les programmes (égalité d'expressions par exemple). La programmation fonctionnelle fournit des moyens de structuration et d'abstraction très puissants :

- fonctions de haut niveau : les comportements qui apparaissent identiques dans des processus différents peuvent être *abstraits*, c'est à dire que l'on peut écrire un processus qui définit ce comportement indépendamment de son utilisation.

- évaluation paresseuse : les expressions sont évaluées seulement lors de leur utilisation effective. Ceci permet la définition et l'utilisation de structures de données infinies.

- abstraction des données : les données comme les comportements peuvent être abstraits sous forme de types de données abstraits.

Capacité d'abstraction

Les fonctions ont le statut "d'objets de première classe", elles peuvent être manipulées sans être nommées, passées en argument, rendues en résultat de procédures et stockées dans des structures de données. Ainsi les comportements fonctionnels peuvent être abstraits sous la forme de fonction de haut niveau. Les données aussi peuvent être abstraites.

Constructibilité

Deux caractéristiques des langages fonctionnels augmentent la constructibilité :

- les fonctions ayant le statut "d'objets de première classe" peuvent être assemblées, "collées", de manière très simple.

- l'évaluation paresseuse permet d'écrire des programmes portant sur des données infinies, impossibles à exprimer dans un langage classique.

Distance téléologique

Les propriétés des expressions (transparence référentielle, forme textuelle explicite...) ont pour conséquence fondamentale qu'une expression représente *toujours* la même valeur. En quelque sorte les expressions peuvent être vues comme des *noms complexes* pour leur valeur. C'est un résultat très important car le processus de programmation consiste alors directement en écriture d'une représentation de la valeur.

Domaines d'application

Par comparaison avec la programmation impérative, il existe une rupture importante dans la manière de penser et de résoudre les problèmes dans un langage fonctionnel, dans le sens d'une réduction de la distance téléologique, même si un langage fonctionnel général n'est pas forcément adapté à tous les domaines d'application comme les systèmes temps-réel et interactifs.

2.3. LA REPRESENTATION DU TEMPS

La notion de temps est associée aux notions d'état et de changement d'état. On se rend compte de l'écoulement du temps en constatant le changement d'état des objets du monde réel. Dans les processus informatiques, le temps intervient :

- de manière interne sous la forme du temps de calcul.
- dans la modélisation ou l'interaction avec des systèmes temporels du monde réel.

Comme nous allons le voir, cette distinction n'est pas toujours évidente à faire dans les différents langages.

Les langages impératifs contiennent une notion d'état et par l'opération d'affectation, gèrent de façon naturelle la notion de changement. La modélisation de processus temporels se réalise alors sous la forme d'objets à état local, l'évolution des objets du monde réel étant simulée par l'évolution des objets informatiques correspondants. Le problème de cette approche est que, comme nous l'avons souligné auparavant, la manipulation d'objets à états, complexifie les raisonnements sur les programmes et leur manipulation

formelle. De plus, le temps de calcul et le temps simulé ne sont pas indépendants. Puisque la séquence des événements temporels simulés est liée à la séquence des instructions du programme, la représentation du temps est *confondue* avec le modèle de calcul sous-jacent du langage.

L'approche fonctionnelle est à priori mal adaptée à la manipulation du temps. En effet, la fonction, objet de base du langage, est *atemporelle* par nature. Une expression représente une seule et même valeur dans le passé, dans le présent et dans le futur ! Différentes techniques ont été développées pour tenter de résoudre ce problème. Un des moyens couramment utilisé pour modéliser des systèmes temporels est de représenter leur évolution sous la forme de *flots d'information*, représentant la suite des états du système. Avec un langage à évaluation paresseuse il est même possible de représenter ainsi des processus infinis. Dans une représentation de ce type, le temps devient explicite dans les données et l'on peut décrire des processus temporels en restant dans le domaine fonctionnel et en gardant donc ses avantages. Bien que cette approche soit une amélioration quant à la représentation du temps, elle n'est pas suffisante dans la perception du temps dont on a besoin dans le domaine musical. Le temps n'est explicite en effet que dans les *données manipulées* par les programmes et non dans les processus eux mêmes.

∴

3. LE PRINCIPE D'HOMOGENEITE

3.1. DEFINITION DU PRINCIPE D'HOMOGENEITE

L'analyse précédente nous amène à formuler le principe suivant :

> *il est possible de réduire la distance téléologique d'un système formel à son domaine d'application en créant une homogénéité conceptuelle entre les deux.*

En d'autres termes, il s'agit de proposer une certaine "ressemblance", une "analogie de forme" entre les formulations du système et les problèmes que l'on souhaite traiter. Dans le domaine de la composition musicale, le principe d'homogénéité doit permettre à l'utilisateur de considérer la formulation d'un processus compositionnel non seulement comme représentative de son résultat, mais également comme participant d'une même essence. S'il est possible de découper, monter et organiser dans le temps des matériaux musicaux, de les représenter sur une partition, il doit être possible de réaliser toutes ces opérations également, et de manière identique, sur des processus compositionnels.

Le principe d'homogénéité doit faciliter le passage de "l'idée" à sa formulation, mais également permettre à une formulation de mieux représenter "l'idée" sous-jacente pour l'utilisateur. Il doit également faciliter le transfert du savoir-faire de l'utilisateur du domaine d'application vers celui de la construction des processus opératoires qu'il souhaite mettre en oeuvre.

Le principe d'homogénéité doit offrir au compositeur un renversement de perspective. Il ne s'agit plus d'envisager l'activité de composition comme une extension de l'activité de programmation, ni le programme informatique comme une représentation de la partition musicale, mais au contraire d'inscrire l'activité de programmation et de description d'opérations compositionnelles dans le prolongement naturel de l'activité de composition.

Dans le domaine musical, la forme la plus apparente est celle de l'organisation temporelle des objets musicaux. Nous nous proposons de conserver cette analogie en introduisant une *dimension temporelle explicite* dans toutes les constructions du langage et non plus seulement dans les données traitées comme cela à été fait jusqu'à présent. Cette approche consiste à traiter le temps en le considérant comme une *dimension* analogue à une dimension spatiale, et par là même de le manipuler dans le "domaine intemporel" des mathématiques.

Introduire une dimension temporelle explicite dans les opérateurs d'un langage impératif ou d'un langage objet paraît être une tâche très difficile. Nous nous sommes donc tournés vers les langages fonctionnels qui offrent:

- une approche par expressions dont le sens est explicite, ce qui facilite la compréhension et le raisonnement sur les programmes, ainsi que leurs transformations formelles.

- des possibilités d'abstraction plus importantes par le biais de fonctions d'ordre supérieur, ce qui facilite la représentation et la manipulation des processus compositionnels en tant que tels.

- une constructibilité plus forte en raison du statut "d'objets de première classe" des fonctions et du mécanisme d'évaluation paresseuse.

- une indépendance de l'ordre d'évaluation des expressions qui favorise l'implémentation sur des machines parallèles.

3.2. UN EXEMPLE DE MISE EN OEUVRE DU PRINCIPE D'HOMOGENEITE : LE G-CALCUL

Le G-Calcul (G pour graphique) que nous présentons ici est un exemple de mise en oeuvre du principe d'homogénéité. Son domaine d'application est la construction et la manipulation d'objets graphiques tri-dimensionnels, et comme nous allons le montrer, toutes les constructions du G-Calcul, y compris les opérateurs et les fonctions, peuvent être "vues" comme des objets graphiques tridimensionnels. Plus précisément, toute construction du G-Calcul est à la fois :

- un objet graphique 3-D qui peut être visualisé sur un écran

- un opérateur 3-D qui peut être combiné avec d'autres opérateurs pour définir des opérations plus complexes .

- un opérande 3-D qui peut être manipulé par d'autres opérateurs.

Dans les parties suivantes, nous allons présenter le G-Calcul en plusieurs étapes:

- nous essayerons en premier lieu de présenter les changements conceptuels dans l'activité de programmation de ce langage à travers une histoire concrète,

- nous donnerons par la suite des exemples plus complexes de construction d'opérateurs,

- nous présenterons une formalisation du langage,

- enfin, nous chercherons à rattacher les concepts développés dans ce langage à la problématique musicale.

3.2.1. Présentation imagée : l'histoire d'un fabriquant de pièces en plastique

Cet exemple a pour but d'illustrer les changements conceptuels dans l'activité de programmation qu'implique le G-Calcul. Nous resterons volontairement dans une représentation très imagée afin d'illustrer les éléments caractéristiques du langage.

Imaginons un fabriquant de pièces en matières plastique qui a besoin de renouveler son matériel de fabrication. Il s'adresse à un fournisseur qui lui propose :

- de la matière première sous la forme de cubes en plastique de couleurs différentes: rouge , blanc , vert , bleu ... etc..

- des moules de différentes formes en plastique transparent,

- une presse à plastique.

Le fournisseur lui explique que ce matériel d'un type nouveau, très performant, a trois caractéristiques particulières:

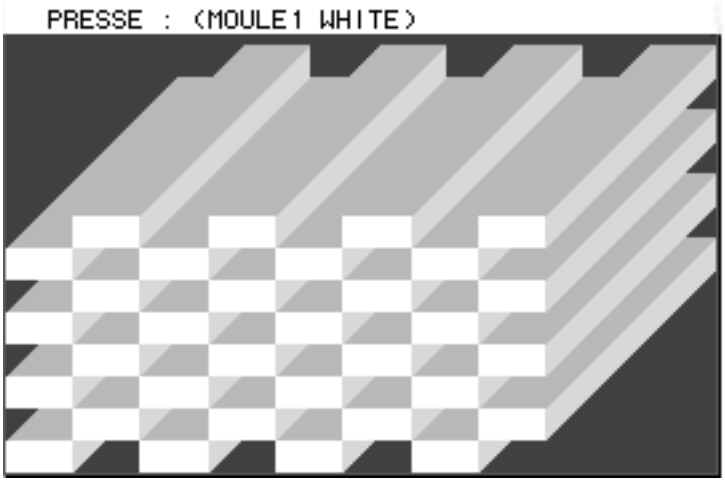
- les moules d'origine ne sont pas en métal comme les moules classiques, ils sont fabriqués à partir de la même matière plastique que les cubes de couleurs, mais ils ont la particularité d'être transparent, ce sont en quelque sorte des moules "abstraits". On ne peut donc appréhender leur forme que lorsqu'on les utilise avec la presse pour former une nouvelle pièce.

- toutes les pièces fabriquées peuvent être utilisées à nouveau en temps que moules sur des cubes de matières colorés. Ceux ci prendront la forme du moule et une couleur résultant du mélange de leur couleur et de la couleur du moule.

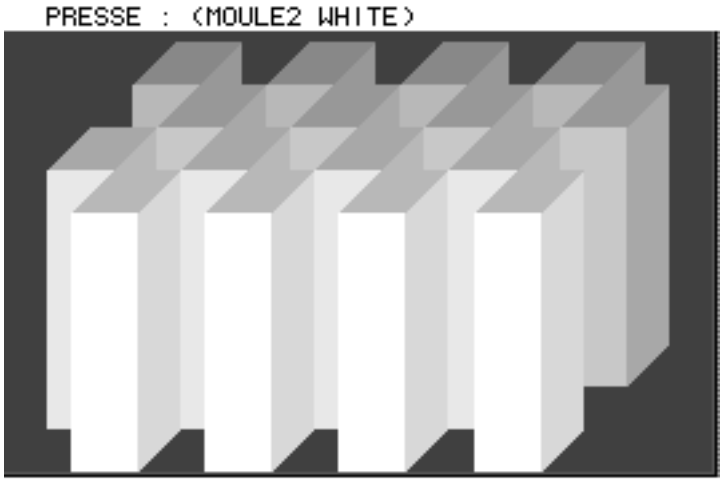
- la presse sera utilisée *de la même manière* pour fabriquer des nouvelles pièces ou des nouveaux moules.

Ainsi lui précise le fournisseur, "vous pourrez fabriquer tous les moules dont vous aurez besoin en utilisant votre propre savoir-faire. Pour construire des moules nouveaux, il vous suffira d'utiliser la presse et les moules de base qui sont fournis ou des moules que vous aurez déjà fabriqués auparavant. Vous obtiendrez alors soit un nouveau moule transparent "abstrait", soit un moule de couleur. Ainsi vous ne serez plus dépendant du fabriquant de moules, vous pourrez construire les moules qui vous conviennent exactement et vous deviendrez plus autonome"

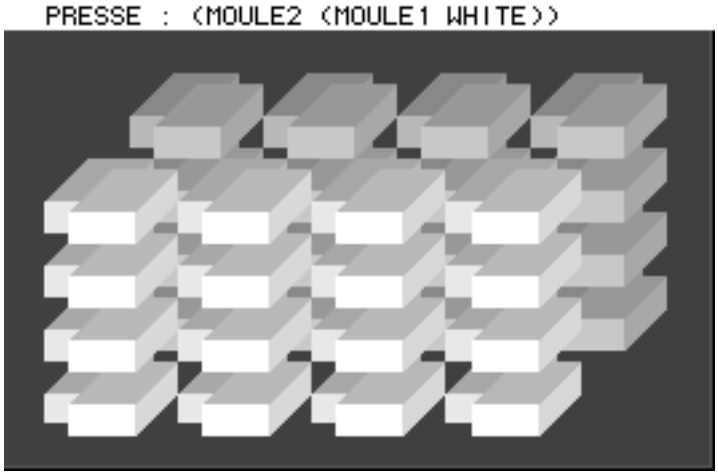
Il lui donne en plus des moules de base et, à titre d'exemple, deux moules complexes transparents déjà construits. Le fabriquant peut alors essayer directement les moules complexes sur un cube de plastique blanc. Il utilise tout d'abord le **moule1** qui lui donne une pièce de la forme suivante:



Avec le **moule2** il obtient une autre forme

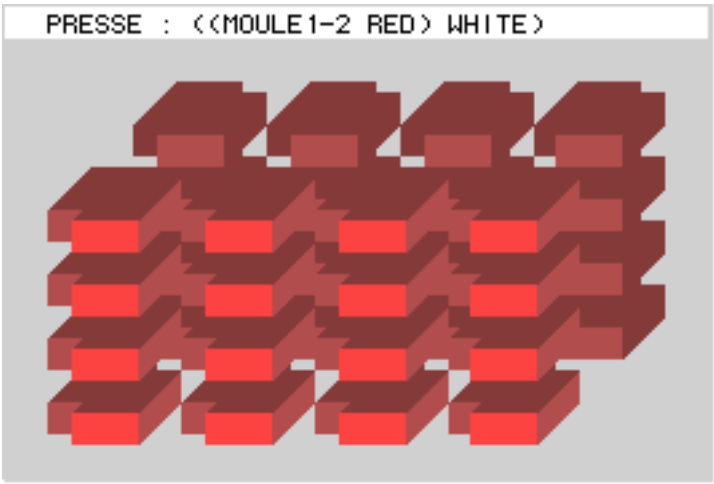


Il peut construire une pièce plus complexe en pressant une copie de la première pièce blanche avec le **moule2** . Il obtient le résultat:



Se rappelant ce que lui a dit le fournisseur, il peut maintenant fabriquer un nouveau moule permettant de presser la nouvelle forme en une seule opération au lieu des deux moulages successifs précédents.

Il définit un nouveau moule transparent **moule1-2= (moule1 moule2)** qui pourra servir à fabriquer une pièce rouge, elle même utilisée comme moule sur un cube blanc (le résultat de couleur rose, mélange du rouge et du blanc est ici représenté par du gris plus foncé).



En quelques opérations, ce fabriquant a construit 4 pièces (utilisables éventuellement comme moules) et un nouveau moule transparent "abstrait" ceci en utilisant seulement la presse, des moules transparents de base ou déjà construits, de la matière première de couleur et son savoir-faire.

Nous pouvons donc dire que pour le fabricant de pièces en plastique :

- un objet, c'est à dire une pièce de plastique transparente ou colorée est soit un résultat, soit un opérateur (un nouveau moule), soit un argument d'une opération de pressage et donc les opérateurs de manipulation et les objets manipulés sont du même type.

- l'activité de construction de nouveaux outils (ici de nouveaux moules) se fait par la même opération (le pressage) qui définit l'activité finale, c'est à dire la fabrication de nouvelles pièces.

Ainsi, la fabrication de nouvelles pièces et la définition de nouveaux outils sous la forme de moules se fait dans un même domaine de travail et à partir de la même opération de pressage.

3.2.2. Présentation détaillée

A la suite de cette première présentation, nous allons montrer comment se définissent des opérations plus complexes en utilisant les opérateurs de constructions de base. Nous avons utilisé dans l'exemple précédent des opérateurs déjà construits (dont on donnait une représentation imagée sous forme de moules). Ces opérateurs ont des dimensions dans le sens où ils ont un *comportement différent* en chaque point de l'espace. Ils sont construits en assemblant des opérateurs primitifs ou déjà définis à l'aide de constructeurs de positionnement dans l'espace.

Les couleurs de base: les opérateurs primitifs

Comme nous l'avons déjà dit plus haut, toute expression du G-Calcul est un espace 3D. Il existe cinq espaces colorés de départ qui sont : rouge, vert, bleu, blanc, noir, et qui seront représentés par un cube de la couleur correspondante.

Les positionnements dans l'espace

Les opérateurs peuvent être positionnés suivant les trois directions de l'espace à l'aide de trois constructeurs :

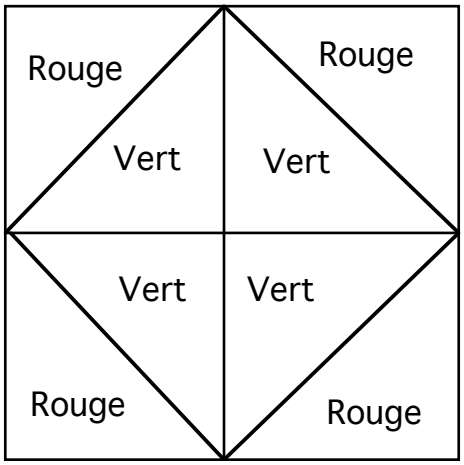
/ positionnement vertical

- positionnement horizontal

$ positionnement dans la profondeur

L'exemple suivant va nous permettre d'illustrer la construction d'un opérateur tridimensionnel **losange**. Lorsqu'il est appliqué à deux couleurs **rouge** et **vert**, il donne le résultat illustré par la figure 1.

-L'opérateur **losange** est construit en plaçant dans l'espace quatre "images" de triangle à deux couleurs donc quatre opérateurs de dessin de triangles :



Il peut être défini de la façon suivante:

losange c1 c2 = (tri1 c1 c2 / tri2 c1 c2) - (tri2 c2 c1 / tri1 c2 c1)))

tri1 et **tri2** sont deux opérateurs récursifs qui dessinent des triangles:

tri1 c1 c2 = (c1 / (tri1 c1 c2) - (tri1 c1 c2) / c2)))

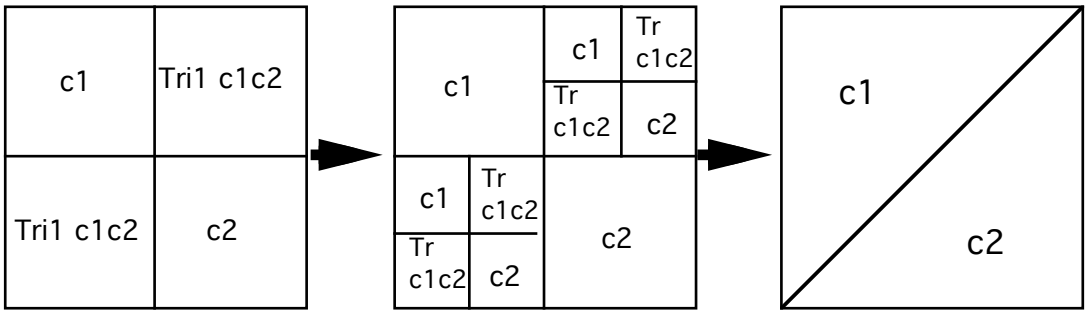
tri2 c1 c2 = (((tri2 c1 c2) / c1) - (c2 / (tri2 c1 c2)))

figure 1 *: (losange rouge vert)*

figure 2 *: (gruyere rouge)*

figure 3 *: (gruyere (losange red green))*

Par exemple **tri1** se représente de la façon suivante:



L'opérateur **tri1** place ses deux couleurs arguments dans les carrés en haut à gauche et en bas à droite, et se place récursivement dans les deux carrés restants.

Un deuxième opérateur **gruyere** est construit par un assemblage récursif complexe d'opérateur **gru** avec :

gru = ((trou / identité) - (identité/ trou)))

où **identité** laisse son argument intact et **trou** efface l'opérateur sur lequel il est appliqué. Appliqué sur la couleur rouge, il donne comme résultat un cube rouge "découpé" (figure 2).

L'opérateur abstrait **gruyere** appliqué à l'image **(losange red green)** donne le résultat visible sur la figure 3, soit un cube rouge contenant un losange vert "découpé" par l'opérateur **gruyère**, ce résultat complexe pouvant être appliqué à d'autres objets ou être manipulé par d'autres opérateurs.

Ces exemples non exhaustifs ont permis de montrer les techniques de construction d'opérateur ayant des dimensions. Ainsi nous visualisons de quelle façon les opérateurs ont un *comportement différent* (et donc donne un résultat différent) en chaque point de l'espace.

3.2.3. Présentation Formelle

La transformation de l'activité de programmation dans le G-Calcul est liée de manière importante à des modifications de la définition opérationnelle du langage. En effet aussi bien les éléments primitifs que l'opération de construction de base (application) ont été définis de manière à formaliser l'idée de dualité expressions/résultats dans un domaine graphique.

Le G-Calcul est un sur-ensemble du Lambda-calcul (voir annexe) dans lequel les expressions acquièrent des *dimensions explicites.* Alors que les expressions du lambda-calcul sont ponctuelles, les expressions du G-Calcul en comportent trois et définissent donc un "espace".

Règles de formation des expressions :

Une expression légale du G-Calcul est formée de la façon suivante:

```
<exp>   ::=     <id>                        // identificateur de variable
                |       (<id> = <exp> ) // abstraction
                |       (<exp> <exp> )  // application
                |       (COLOR c )              // couleur de base
                |       (<exp>  / <exp> )        // positionement gauche droite
                |       (<exp>  - <exp> )        // positionnement haut bas
                |       (<exp>  $ <exp> )        // positionnement avant arrière
                |       (LEFT <exp>)   // découpe partie gauche
                |       (RIGHT <exp>)// découpe partie droite
                |       (TOP <exp>)              // découpe partie haute
                |       (BOT <exp>)              // découpe partie basse
                |       (FORE <exp>)   // découpe partie avant
                |       (BACK <exp>)   // découpe partie arrière

<id>            ::=     <lettre>
                |       <lettre><id>

<lettre> ::=    a | b | c ...
```

En d'autres termes, une expression du G-Calcul est soit:

- une expressions classique du Lambda-calcul
- une couleur de base

- une expression "construite" à l'aide des trois opérateurs de positionnement dans les trois directions de l'espace, / , - , $

- une expression formée en utilisant les "destructeurs" (Left, Right, etc ...)

L'application

L'application est dans le Lambda-calcul l'opération qui formalise l'idée de *constructibilité* du langage. En effet l'opération d'application permet de construire des nouveaux opérateurs en combinant des opérateurs primitifs ou déjà construits.

Dans le G-Calcul l'évaluation d'une expression se fait de la façon suivante:

- lorsqu'une couleur est appliquée sur une autre couleur, le résultat est le mélange des deux.

- lorsqu'une abstraction est appliquée sur ses arguments, ceux-ci sont distribués à "l'intérieur" de l'opérateur par le mécanisme habituel de substitution paramètres formels arguments.

- lorsqu'un opérateur structuré est appliqué à une autre opérateur, l'application se "distribue" par dimension. Ainsi par exemple la moitié gauche (respectivement droite) de l'opérateur s'applique sur la moitié gauche (respectivement droite) de l'opérande, (de la même manière pour les autres dimensions) et ce mécanisme se répète récursivement sur les deux moitiés. Ainsi chaque point de l'opérateur s'applique sur le point correspondant de l'opérande.

On peut écrire l'opération d'application d'une expression construite verticalement sur un opérande de la façon suivante:

(e1 / e2) @ e3 ==> (e1 @ (left e3) / e2 @ (right e3))

Cette règle de réécriture étant transposable dans les autres dimensions.

Cette distribution de l'application dans l'opérateur à pour conséquence que plusieurs applications se déroulent dans des directions différentes. L'évaluation se fait en parallèle dans des sous-espaces différents et s'arrête en un point donné :

- si le résultat est une couleur,

- si le sous-espace évalué ne peut plus être dessiné entièrement sur l'écran. Dans ce cas c'est l'une ou l'autre des branches (gauche ou droite, haute ou basse ...etc..) qui sera évaluée pour rendre un résultat.

∴

4. VERS UN ESPACE DE COMPOSITION HOMOGENE

Nous proposons d'esquisser ici les grandes lignes d'un "espace" de composition musicale basé sur le principe d'homogénéité précédemment décrit, permettant d'étendre largement le cadre de l'écriture et de la notation musicale, tout en s'inscrivant dans la continuité des pratiques et des savoir faire actuels. Nous présenterons tout d'abord les bases théoriques d'un langage de composition musicale homogène. Nous verrons ensuite sa mise en oeuvre dans le cadre d'un environnement de travail spécifique. Nous nous intéresserons également au problème de la conservation et de la pérennité des documents musicaux ainsi crées.

4.1. LES BASES D'UN LANGAGE DE COMPOSITION HOMOGENE : LE T-CALCUL

Le T-Calcul, dont nous donnons ici une présentation qui n'est pas encore complètement formalisée, fournit les bases théoriques d'un nouveau type de langage de composition musicale mettant en oeuvre le principe d'homogénéité. Il a pour objectif la manipulation et la construction d'objets ayant une dimension temporelle explicite, tout en restant dans le domaine purement fonctionnel de façon à bénéficier des avantages de ce type de programmation :

a) Absence d'états et d'effets de bord

b) Transparence référentielle

c) Indépendance de l'ordre d'évaluation

d) Abstractions de haut niveau

e) Evaluation "paresseuse"

Les objets manipulés par le T-Calcul, les *T-expressions*, peuvent être vues comme des "partitions" décrivant l'organisation dans le temps de λ-expressions. Une T-expression peut être appliquée à une autre par une opération *d'application temporelle*. Le résultat est une nouvelle T-expression résultant des applications individuelles de chaque composante.

4.1.1. Analogies avec le G-Calcul

Nous allons proposer une analogie entre l'histoire concrète développée auparavant pour le G-Calcul et ce que permettra le T-calcul. Celui-ci manipule des objets ayant une dimension temporelle au lieu de dimensions

spatiales. Pour cela, nous allons résumer les différentes notions importantes dans un tableau.

	Fabricant	**Compositeur**
Activité finale du langage	Fabrication de pièces colorées	Composition de matériaux passifs et de processus actifs
Activité de programmation	Fabrication de moules	Création de nouveaux processus compositionnels
Savoir-faire	Opération de pressage	Opération de composition : arrangement de matériaux dans le temps

L'histoire du fabricant de pièces en plastique nous a permis de montrer que:

- tous les objets manipulés sont du même type : des pièces en matière plastique,

- la fabrication de nouveaux outils sous la forme de moules (l'activité de programmation) et la fabrication de pièces (l'activité finale) se réalise de la même façon par l'opération de pressage, c'est à dire avec un savoir-faire unique.

De façon similaire et dans le contexte musical, le T-calcul doit permettre :

- la manipulation d'objets d'un seul type, des partitions musicales.

- la définition et la manipulation de nouveaux processus compositionnels à partir du savoir-faire habituel du compositeur que l'on peut résumer sous la forme de l'opération de composition.

Ainsi, le T-Calcul doit permettre de composer, de représenter et de manipuler des programmes et des processus compositionnels comme l'on compose, représente et manipule des matériaux musicaux traditionnels ceci dans un domaine totalement unifié où les matériaux passifs et les opérateurs actifs seront représentés de la même manière.

4.1.2. T-expression

Une T-expression (une expression du T-Calcul) est une fonction du temps associant à chaque instant un ensemble, éventuellement vide, de λ-expressions. Ainsi T_1, la T-expression de la figure 1 associe :

t0	∅	{ E1 }
t1	∅	{ E2 }
t2	∅	{ E2, E3 }
t3	∅	{ E3 }
t4	∅	{ E4 }
t5	∅	{}

ou E1, E2, E3, E4 sont des λ-expressions. Nous noterons T[t] l'ensemble des valeurs d'une T-expression T à l'instant t.

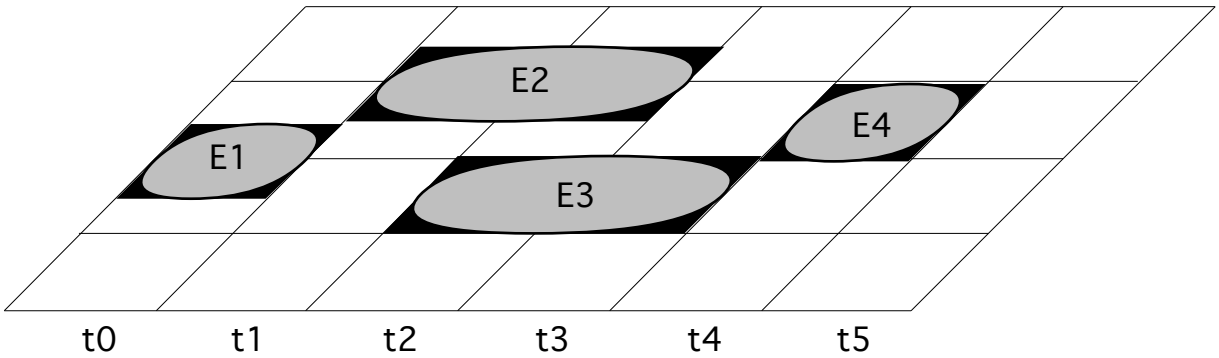


Figure 1 : T_1

4.1.3. Application Temporelle

L'application temporelle, que nous noterons $@_T$, est une opération binaire analogue à l'application du Lambda-Calcul, mais portant sur des T-expressions.

Ainsi, soit deux T-expressions T_1 et T_2 :

$$(T_1 @_T T_2)[t] \varnothing \qquad \{ (E_i\ E_j) \ / \ \forall\ E_i \in T_1[t] \text{ et } \forall\ E_j \in T_2[t] \}$$

ou $(E_i\ E_j)$ indique l'application de E_i sur E_j.

En d'autres termes le résultat de l'application temporelle de T_1 sur T_2 est une T-expression associant à chaque instant t l'ensemble constitué des applications de chaque λ-expression de T_1 à l'instant t sur chaque λ-expression de T_2 à l'instant t.

Le tableau 1 ci-dessous détaille le résultat de l'application temporelle $(T_1 @_T T_2)$.

	T1	**T2**	**(T1 $@_T$ T2)**
t0	{ E1 }	{ E5 }	{ (E1 E5) }
t1	{ E2 }	{ E5 }	{ (E2 E5) }
t2	{ E2, E3 }	{ E5, E6 }	{ (E2 E5), (E2 E6), (E3 E5), (E3 E6) }
t3	{ E3 }	{ E5, E6 }	{ (E3 E5), (E3 E6) }
t4	{ E4 }	{ E6 }	{ (E4 E6) }
t5	{}	{ E6 }	{}

Tableau 1 *: (T_1 $@_T$ T_2)*

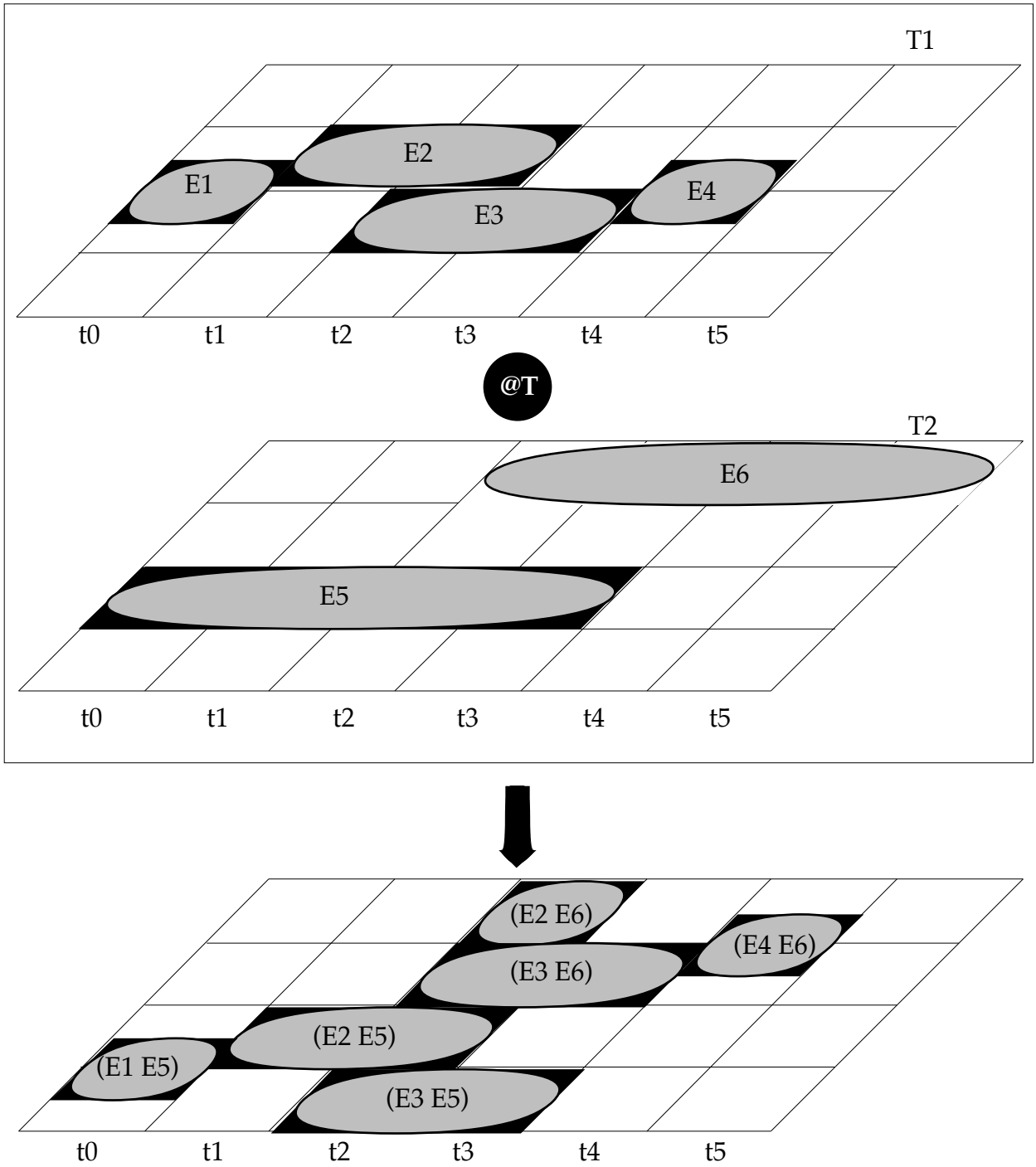


Figure 2 *: (T_1 @$_T$ T_2)*

4.2. DEFINITION D'UN ESPACE DE COMPOSITION MUSICALE.

Cet espace vise à proposer au compositeur une métaphore[1] de travail intuitive et cohérente, prenant en compte, grâce au T-Calcul, aussi bien les matériaux musicaux que les processus compositionnels mis en oeuvre. Cette métaphore s'appuie sur cinq éléments clefs :

- Un concept d'objet musical généralisé pouvant recouvrir à la fois des objets "passifs", les matériaux musicaux et des objets "actifs", les processus de composition.

- Un "espace de composition" unique sur lequel le compositeur peut graphiquement créer, noter, placer et organiser dans le temps tous les objets musicaux, c'est-à-dire aussi bien les matériaux musicaux que les processus de composition.

- Des opérateurs de composition, de hiérarchisation et d'abstraction pouvant être utilisés indifféremment pour former de nouveaux matériaux musicaux ou de nouveaux processus compositionnels.

- Une dimension "historique" des objets musicaux qui conservent intégralement la description de leurs étapes constitutives.

- Une équivalence totale entre les différentes représentations : graphiques, textuelles, etc.

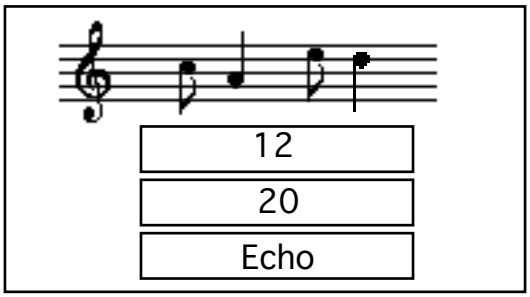
Nous pouvons esquisser quelques-unes des conséquences de cette approche[2]:

• L'application d'une fonction de transformation à une structure musicale se fera par simple "juxtaposition" graphique des deux objets sur l'espace de composition.

---

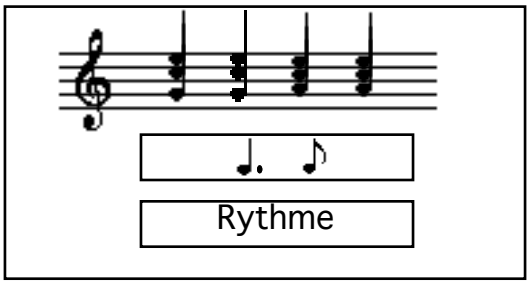
[1] La métaphore qui sous-tend une interface utilisateur permet à celui-ci, par un transfert de sens, de développer un modèle cohérent du fonctionnement d'une application. Si elle est bien choisie, l'utilisateur peut plus facilement prévoir le résultat d'une action ou deviner quelle action entreprendre pour obtenir le résultat souhaité. Un exemple désormais classique de métaphore est celle du "bureau" du Macintosh.

[2] Les illustrations qui suivent sont données à titre d'exemple, sans préjuger de la forme finale qui sera adoptée.

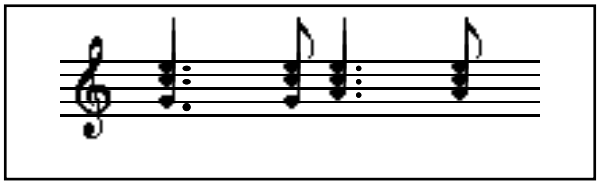


L'application de l'opérateur "écho" se fait par empilement graphique de l'objet le représentant et de ses différents arguments (un délai : 20, son amortissement : 12, et une séquence de notes).

• Chaque objet musical ainsi créé porte en lui l'historique de son processus de composition. Il peut être indifféremment représenté dans l'espace de composition comme "histoire" :



ou comme "résultat" :

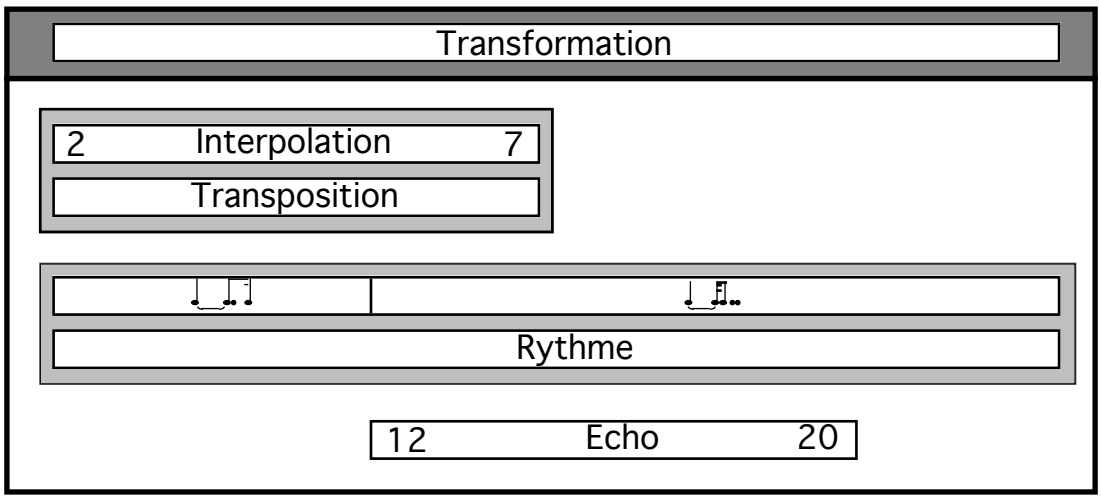


suivant ce que veut faire apparaître l'utilisateur.

• Cette "histoire" peut être à tout moment "révisée" sans que l'utilisateur n'ait à se rappeler ni à reformuler toutes les étapes de travail : elle est toujours intrinsèquement présente dans l'objet musical, on peut accéder à sa représentation qui se prête aisément à des mécanismes d'édition tels que 'supprimer', 'remplacer', 'modifier'.

• Elle peut également être "abstraite" pour en extraire le processus de composition sous-jacent et enrichir les fonctionnalités du langage. Ainsi, il n'est plus demandé au compositeur de programmer pour composer. Au

contraire, la programmation de nouvelles fonctions du langage sera vue comme une extension de l'activité de composition.



L'opérateur "Transformation", vu ici à la fois comme "histoire" et comme "résultat", est composé graphiquement d'un opérateur "Transposition" prenant comme argument le résultat d'un opérateur "Interpolation", d'un opérateur "Rythme" prenant deux arguments successifs dans le temps, et d'un opérateur "Echo". Tous ces opérateurs sont combinés dans le temps par leurs positions relatives, leur durée d'application étant proportionnelle à leur longueur. Le résultat de cette composition d'opérateurs est donc un nouvel opérateur : "Transformation", qui prendra un argument : la séquence de notes à laquelle il va s'appliquer.

Ainsi, l'extension du langage du compositeur sera vue par celui-ci comme l'extension de son système de notation, rejoignant en cela des pratiques devenues courantes en musique contemporaine.

4.3. DEFINITION D'UN NOUVEAU TYPE DE DOCUMENT MUSICAL.

Dans le contexte de notre recherche, le document sur lequel va travailler le musicien doit se soumettre aux exigences de la partition traditionnelle, bien que le cadre de la notation y soit considérablement élargi : les matériaux avoisinent les processus, son contenu doit se prêter tant à la lecture de l'interprète, du compositeur que de la machine. Le document musical doit donc s'appuyer sur :

- une structuration descriptive, logique et temporelle de ses différents éléments qui permette de le rendre indépendant de son support matériel et logiciel.

- la représentation d'éléments procéduraux ou fonctionnels[3]
- la multiplicité de ses représentations : il doit permettre de faire apparaître tant son contenu musical (l'oeuvre comme résultat) que les processus de composition (l'oeuvre comme histoire).

Notre ambition première, dans cette tentative de redéfinition du document musical, est de permettre sa restitution indépendamment de tout support matériel ou logiciel et de garantir ainsi sa pérennité.

4.3.1. La nécessité d'une normalisation du document musical.

Il est difficile d'imaginer que les évolutions technologiques nous conduiront un jour à l'établissement d'une plate-forme matérielle stable. Parce qu'ils leur sont étroitement liés et même s'ils semblent graviter autour de quelques grands standards, les systèmes d'exploitations sont également en constante évolution. L'industrie naissante du logiciel doit dans ce contexte, s'affirmer comme telle à part entière, distincte des industries matérielles et systèmes, et doit protéger ses investissements contre les changements technologiques. C'est dans ces préoccupations, d'ordre général, qu'il convient de replacer les rapports de la musique à l'informatique.

Une solution permettant de préserver l'intégrité de données dans le temps et sur des supports matériels différents est la normalisation de ces données. Le domaine musical est riche de normes qui malgré leurs limites, ont fait la preuve de leur efficacité : la norme MIDI et sa dérivée, la norme MIDIFILE. La première permet l'échange de données entre des supports matériels totalement hétérogènes ainsi que leur traitement informatique, la seconde permet la conservation de ces données dans le temps ainsi que leur accès via des supports logiciels différents. Elles présentent cependant bon nombre d'insuffisances qui les rendent inadéquates pour traiter un document musical dans toute sa généralité : elles ne permettent pas, par exemple, de décrire d'une manière standard, la matière sonore manipulée (celle-ci est dépendante de la machine réceptrice de l'information), elles ne permettent nullement de garder trace du processus génératif de l'oeuvre musicale ...

---

[3] La structure du document musical va s'appuyer sur un langage de description normalisé. La représentation d'éléments procédurals ou fonctionnels signifie en quelque sorte que ce langage devra permettre la description d'autres langages, de manière indépendante de leur implémentation.

Néanmoins, c'est très certainement sur la voie de la normalisation des documents musicaux (le terme de document musical devant être compris ici comme général, englobant la matière sonore, les événements musicaux, les processus compositionnels ainsi que leur ordonnancement dans le temps), que la partition informatique pourra se dégager de sa dépendance vis à vis de son support matériel et logiciel.

4.3.2. Les bases potentielles d'une normalisation.

Le problème de normalisation et d'échange de documents est actuellement une préoccupation d'ordre général de l'industrie du logiciel. "Une solution adaptée à l'échange de documents, prenant en compte le texte, les graphiques et l'image, ne peut provenir que d'une structuration de ces documents, qui soit indépendante des systèmes d'exploitation et des plates-formes matérielles. Il est nécessaire de définir une architecture qui transcende les limites de l'ordinateur, pour développer l'indépendance et l'exhaustivité de la structure de l'information."[4]

En réponse à ces problèmes de traitement et d'échange d'information, différents travaux ont été menés par l'International Standardisation Organisation (ISO), qui ont abouti à la définition d'une première norme de structuration de documents : SGML (1986), puis en Mai 1992, à l'adoption comme Draft International Standard, d'une seconde norme, HyTime, basée sur SGML, élargie au traitement de documents multimédias et prenant en compte l'espace et le temps pour la restitution de ces documents. Aujourd'hui, et toujours dérivée des deux premières, une troisième norme est en cours d'élaboration : SMDL, prenant en compte cette fois la spécificité des documents musicaux.

ISO 8879 - 1986 : Standard Generalized Markup Language (SGML).

SGML est un langage de marquage généralisé de documents. Au marquage procédural, qui associe une procédure de formattage, par exemple, à une portion de texte, SGML oppose un marquage descriptif, destiné à expliciter la structure logique d'un document.

SGML fournit :

---

[4] John E. Warnock - The new age of documents - Byte, June 92.

- une métasyntaxe permettant d'exprimer la syntaxe de différents types de document : le définition de chacune de ces syntaxes se fait grâce à un "document type définition (DTD)" et consiste en un jeu d'identificateurs génériques (étiquettes) permettant d'encoder un type particulier de document, le liant à son contenu, ainsi qu'un jeu d'attributs pour chaque élément générique.

- une métasyntaxe permettant d'exprimer la syntaxe du codage générique dans le document lui-même.

SGML permet à un même document d'apparaître de manière transparente sur différents systèmes hétérogènes. De plus, tous les documents SGML se prêtent à un certain nombre de traitements généraux, parmi lesquels l'interrogation et la consultation.

SGML a été adopté par un certain nombre d'organismes dont le Département de la Défense des Etats-Unis, le parlement Européen, l'industrie aéronautique... D'une manière générale, l'intérêt de SGML comme outil de représentation de l'information est d'autant plus grand que le corps de cette information est grand, complexe et nécessite d'être maintenu dans le temps.

ISO DIS 10744 : Hypermedia/Time-based Structuring Language (HyTime)

Lors de la création d'un document, la distinction doit être faite entre l'information et les instructions de restitution. Cette distinction est aisée pour des documents traditionnels où les données consistent généralement en mots imprimables et ponctuations. Dans le cas de documents multimédia, hypermédia ou temporels, cette distinction est moins évidente. Pour de tels documents, la manière de restituer l'information est souvent critique pour leur compréhension. Seul l'auteur ou le compositeur peut savoir quelles sont les informations de restitution essentielles et quelles sont celles qui peuvent être laissées à l'interprétation d'une "feuille de style". HyTime permet à l'auteur de contrôler cette restitution par un jeu d'instructions qui fait partie intégrante de son travail.

HyTime est un langage permettant de décrire la structure de documents hypertexte, hypermédia, multimédia, de documents spatiaux et temporels. HyTime, dérivé de SGML, est donc interprétable comme tel.

HyTime fournit donc des moyens standards d'exprimer :

- des positions dans ou en dehors de documents, ainsi que l'établissement de liens arbitraires entre celles-ci.

- l'ordonnancement d'objets multimédia dans des espaces de coordonnées finis.

- des instructions de restitution pour des projections arbitraires d'objets dans différents espaces finis ou constructions.

Le modèle de traitement conventionnel d'un document HyTime est le suivant :

- l'application qui souhaite traiter le document fait appel au moteur HyTime qui à son tour lance l'analyseur SGML. Pendant la phase de traitement du document, l'analyseur renvoie au moteur HyTime, toutes les informations qu'il rencontre. Dans le même temps, le moteur HyTime effectue deux opérations :

  • il repasse la sortie du document à l'application qui est libre de la traiter ou de l'ignorer.

  • il crée des structures de données internes, basées sur les informations spécifiquement HyTime retournées par l'analyseur.

- une fois la phase d'analyse terminée, l'application peut alors interroger le moteur HyTime de différentes manières, il assume la responsabilité de l'ordonnancement des différents objets dans l'espace, la conversion de l'adresse d'éléments en données, etc...

Une des fonctionnalités remarquable de HyTime est sa capacité à supporter des informations musicales, de manière à pouvoir les intégrer totalement à d'autres types d'information.

ISO Committee Draft 10743 : Standard Music Description Language (SMDL)

SMDL est une norme en cours d'élaboration qui hérite de SGML et de HyTime. Son groupe de travail (X3V1.8M) s'est réuni pour la première fois en juillet 1986. Son but était d'appréhender tout à la fois le type d'informations présentes dans un flot de données MIDI, ainsi que celles présentes dans une partition musicale.

N'ayant à l'heure actuelle fait l'objet d'aucune adoption en tant que standard, il n'existe pas de publication officielle concernant SMDL.

4.3.3. Les perspectives de normalisation

Sans préjuger de la teneur des travaux actuels sur SMDL, la convergence d'intérêts qui se dégage des normalisations effectuées ou en cours, et certains des besoins de la musique contemporaine est remarquable. Il s'agit dans tous les cas, de préserver dans le temps un travail coûteux pour les uns en terme d'investissement, précieux pour nous en terme de culture musicale. Assurer la pérennité d'un ensemble de documents (y compris le document musical), suppose de permettre sa restitution indépendamment de son support matériel et logiciel, et c'est bien en ces termes que se pose le problème de la migration de la partition traditionnelle vers des supports informatiques. C'est donc sur la voie de ces normalisations qu'il faut envisager l'élaboration de documents musicaux aptes à enrichir durablement le patrimoine musical.

Néanmoins, il est fort probable que les normes parues et en cours ne soient pas suffisantes pour l'ensemble de nos besoins : la définition de sur-couches, la participation éventuelle aux groupes de travail de l'ISO pour préciser, orienter ou compléter ces normes, seront certainement nécessaires.

4.4. ARCHITECTURE GENERALE

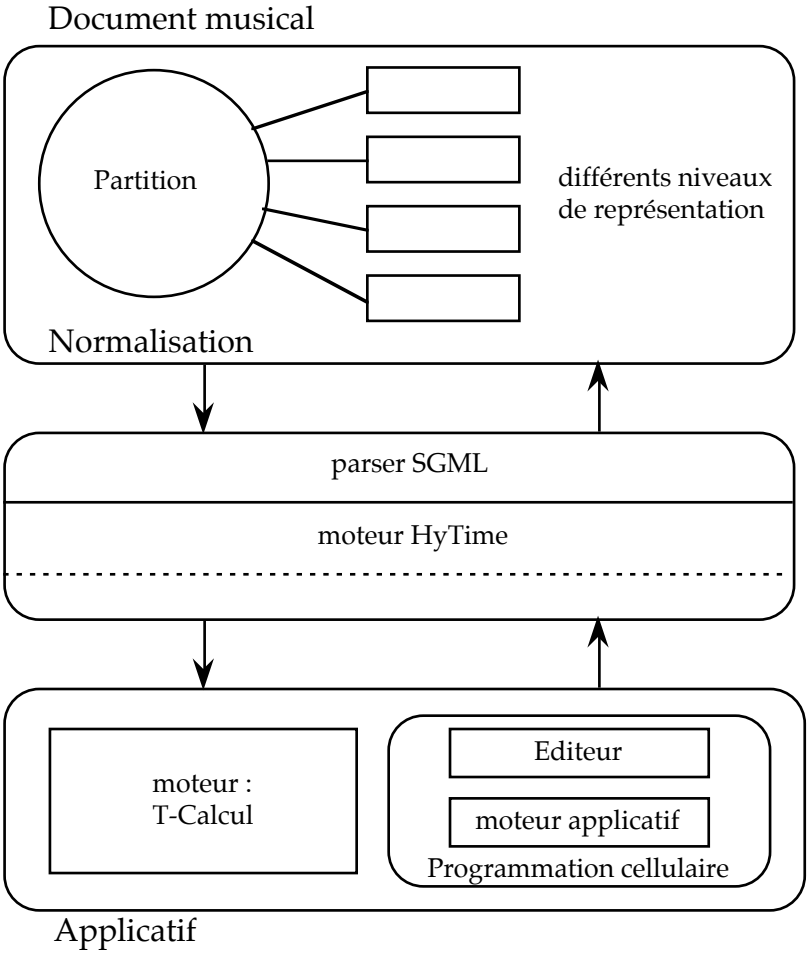
Le système en cours d'élaboration repose donc avant tout sur une approche fonctionnelle des langages de programmation. L'accès aux primitives du langage se fera grâce à des outils graphiques (éditeurs) élaborés selon un modèle de programmation cellulaire[5].

La normalisation des documents musicaux nécessitera quant à elle et au minimum, un module comprenant un analyseur SGML ainsi qu'un moteur HyTime.

---

[5] la programmation cellulaire est un nouveau modèle de programmation que nous sommes en train de développer, mais qui n'est pas décrit ici.

Architecture générale



L'architecture générale du système est donc totalement modulaire : elle devra permettre d'exploiter les fonctionnalités du langage indépendamment de l'applicatif et d'établir des documents musicaux qui ne soient liés ni à la machine, ni au logiciel qui les a créés.

∴

5. CONCLUSION

L'informatique a révolutionné le domaine musical, elle a bouleversé nombre de schémas établis mais s'est malheureusement toujours située en rupture avec les modes de pensée et d'écriture musicaux, limitant en cela son utilisation à un ghetto d'aventuriers technologiques. A l'heure où l'informatique est suffisamment mature pour nous ouvrir l'ensemble du champ des possibles, il est plus qu'important de combler le fossé qui sépare informaticiens et musiciens, afin de permettre l'exploration de cet univers nouveau par le biais de stratégies qui soient en continuité des modes de pensée et d'écriture traditionnels.

L'accès à cet univers se fait par la programmation de la machine, seul mode d'utilisation qui ne soit pas figé dans un ensemble de fonctionnalités fini. Nous pensons que l'approche fonctionnelle des langages de programmation est à l'heure actuelle, la plus adaptée aux problèmes soulevés dans le domaine musical. Nous avons montré également avec le G-Calcul que l'on peut intégrer la notion de dimension au sein même d'un langage et que celle temporelle, indissociable du domaine musical, pouvait de la même manière, servir de base au T-Calcul. C'est ainsi que la représentation d'un opérateur du langage ou d'un processus de composition (en fait, la même entité mais vue sous deux angles diamétralement opposés), pourront se confondre dans la même vue explicite du temps linéaire de la partition. Le musicien pourra alors programmer comme il compose et la pensée musicale, bénéficier de l'ensemble de la pensée algorithmique de l'informatique, ne faisant en cela que rejoindre cette union des sciences et de la musique dont nous héritons de l'antiquité grecque.

Enfin, nous avons ébauché une solution au problème de la partition informatique à travers la constitution et la normalisation d'un document musical plus général, avec le fervent espoir qu'il vienne à enrichir à terme le patrimoine musical avec la même pérennité que la partition traditionnelle. De plus, outre qu'il permettra la conservation et la restitution des oeuvres, le document musical sera porteur d'un nouveau type d'information : la démarche compositionnelle qui a donné naissance à l'oeuvre. Le patrimoine pourra prendre alors une dimension supplémentaire : celle d'une pensée musicale directement interrogeable et accessible à l'analyse.

Sur l'ensemble de ces thèmes, un certain nombre de travaux s'imposent pour prolonger ceux déjà réalisés : il s'agit notamment de finaliser l'implémentation du T-Calcul, de poursuivre les réalisations déjà entreprises pour la mise en oeuvre logicielle des concepts définissant un nouvel espace de composition musicale ainsi qu'à plus long terme, de poursuivre les recherches sur la normalisation du document musical.

Les perspectives de notre démarche se situent dans le prolongement et dans l'extension de la pensée musicale par le biais notamment de la notation généralisée des processus compositionnels. La pratique musicale contemporaine dont la quête se traduit aujourd'hui par l'éclatement des tendances, des formes et des systèmes, y trouvera, nous l'espérons, des moyens d'expression à la mesure de ses ambitions.

∴

6. ANNEXES

6.1. LE LAMBDA CALCUL

6.1.1. Présentation

Le Lambda Calcul est un calcul formel dû au logicien et philosophe américain Alonzo Church (Church, 1941). Il porte essentiellement sur la construction et la manipulation formelle de fonctions au sens mathématique du terme. Son importance théorique et son élégance tiennent au petit nombre d'éléments sur lesquels il se fonde, mais qui lui permettent cependant d'exprimer toute fonction calculable.

Le Lambda Calcul ne contient en effet quasiment aucune des notions habituelle de l'informatique : ni récursivité, ni structures de contrôle (alternative, séquence, répétition,...), ni affectation, ni structures de données. Néanmoins, par construction purement formelle, la récursivité, ainsi que les principales structures de données et les structures de contrôle, peuvent être introduites.

De ce fait, le Lambda Calcul constitue l'un des fondements théoriques de l'informatique en particulier des langages de programmation. Le langage LISP, dans sa version originale, s'en inspire très directement. Il en va de même pour de nombreux travaux actuels sur la programmation fonctionnelle.

6.1.2. Formation des expressions du Lambda Calcul

Le lambda calcul s'appuie sur deux notions : l'*application* et l'*abstraction*.

L'application

L'application est une opération binaire qui consiste à "appliquer" un opérateur à un opérande. L'opérateur agit sur l'opérande pour produire un résultat. Voici quelques exemples de notations ("f" représente l'opérateur et "a" l'opérande) :

a) [f, a] notation de von Neumann

b) f(a) notation usuelle en mathématique

c) (f a) Lambda Calcul

Le Lambda Calcul manipule des opérateurs unaires. Les opérateurs n-aires sont transformés en opérateurs unaires par une opération dite de *curryfication*. Ainsi l'addition de deux nombres par un opérateur curryfié **add** sera notée :

((add 10) 20)

L'expression **(add 10)** est d'abord calculée en appliquant l'opérateur **add** à 10. Le résultat obtenu (un nouvel opérateur "qui ajoute 10") est ensuite appliqué à 20, pour donner 30.

Plus généralement, l'application se note par juxtaposition de l'opérateur et de l'opérande entre parenthèses :

(<exp1> <exp2>)

L'abstraction.

L'abstraction est utilisée pour définir une fonction mettant en relation un paramètre formel et une expression qui généralement en dépend. Voici quelques exemples d'abstractions :

a) $f(x) \equiv x^2 - 3x$

b) $x \varnothing (x^2 - 3x)$

c) $(\lambda\, x\ (x^2 - 3x))$

Le Lambda Calcul utilise la dernière formulation. Plus généralement une Lambda-abstraction s'écrira :

(λ <id> <exp>)

où **<id>** est le nom du paramètre formel de la fonction et **<exp>** le corps de la fonction. Par exemple, pour définir une fonction qui ajoute un nombre à lui-même on pourra écrire :

(λ x ((add x) x))

Règles syntaxiques du Lambda Calcul

La syntaxe du Lambda Calcul est très simple. Les règles de formation des λ-expressions sont résumées ci-dessous.

<exp>	::=	**<id>**	// identificateur de variable
	|	**(λ <id> <exp>)**	// abstraction
	|	**(<exp> <exp>)**	// application
<id>	::=	**<lettre>**	
	|	**<lettre><id>**	
<lettre>	::=	**a | b | c** ...	

Tableau 1 *: Règles syntaxiques du Lambda Calcul*

Pour alléger l'écriture d'expressions complexes, on convient généralement des simplifications suivantes :

a) Associativité de gauche à droite de l'application.

L'expression : **(((<exp1> <exp2>) <exp3>) <exp4>)**

pourra donc être notée : **(<exp1> <exp2> <exp3> <exp4>)**.

De même, l'expression : **((add 10) 20)** pourra être notée : **(add 10 20)**.

b) Associativité de droite à gauche de l'abstraction.

L'expression : **(λ <id1> (λ <id2> (λ<id3> <exp>)))**

pourra donc être noté : **(λ <id1> λ <id2> λ<id3> <exp>)** .

On convient également de pouvoir placer un point (.) entre l'identificateur et l'expression pour mieux les séparer visuellement.

On écrira par exemple : **(λ x.λ y.λz.<exp>)**

6.1.3. Règles de réduction des expressions

Les règles de réduction du Lambda Calcul décrivent comment transformer une lambda-expression dans le but de la réduire (de la simplifier). Si l'on se place d'un point de vue informatique, on va chercher à calculer le résultat d'une lambda-expression en appliquant itérativement les règles de réductions. Quand une expression ne peut plus être réduite elle est dite en *forme normale*.

β-réduction

La principale règle de réduction est appelée β-réduction. Elle consiste à transformer une expression de la forme :

(λ<id> <exp1>) <exp2>

en une copie de **<exp1>** où toute les occurrences libres de **<id>** sont remplacées par **<exp2>.**

Voici un exemple où plusieurs β-réductions sont appliquées successivement à une expression:

(λx.λy.λc.(c x y)) <exp1> <exp2> (λa.λb.a)	**∅β**	**(λy.λc.(c <exp1> y)) <exp2> (λa.λb.a)**
	∅β	**(λc.(c <exp1> <exp2>)) (λa.λb.a)**
	∅β	**(λa.λb.a) <exp1> <exp2>**
	∅β	**(λb.<exp1>) <exp2>**
	∅β	**<exp1>**

La règle de β-réduction n'est en fait pas aussi simple qu'il n'y parait et des précautions doivent être prises dans la substitution pour ne pas créer des conflits entre les noms des variables. En particulier aucune des variables liés du corps de l'abstraction ne doivent être des variables libres de l'opérande. Si tel n'est pas le cas, il convient de renommer ces variables du corps de l'abstraction. On utilise pour cela l'α-conversion.

α-conversion

Cette opération consiste à renommer le paramètre formel d'une abstraction en s'assurant que le nouveau nom n'apparait pas déjà dans le corps de l'abstraction. Voici quelques exemples :

λx.(add x x)	**∅α**	**λy.(add y y)**
λf.(λx.f (x x) λx.f (x x))	**∅α**	**λg.(λx.g (x x) λx.g(x x))**

6.2. IMPLEMENTATION COMMON LISP DU G-CALCUL

```
;************************************************************************************
;************************************************************************************
;                                      G-CALCUL (3D)
;
;                                      Y0 - GRAME
;
; Historique :
;  25-06-92 : première version
;  26-06-92 : résolu pb approximations, ajout constructeur BLEND
;  26-06-92 : ajout destructurations TOP BOT LEFT RIGHT
;  26-06-92 : version 3D : ajout constructeur $ et destructeurs FORE et BACK
;  01-07-92 : modification perspective
;
; Bugs et limitations :
;
;************************************************************************************
;************************************************************************************


(require 'quickdraw)
(defvar *g-env* nil)
(defvar *g-wind* nil)
(defvar *g-w1* nil)
(defvar *g-w2* nil)
(defvar *g-w3* nil)
(defvar *g-w4* nil)
(defvar *g-count* 0)

#|
/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
=====================================================================================
\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/
                                       FORMALISATION

/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
=====================================================================================
\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/


=====================================================================================
DESCRIPTION DES STRUCTURES DE DONNÉES
=====================================================================================
e : expression,
s : symbole,
p : environnement,
c : couleur,
r : rectangle.


EXP ::= <COLOR c>, <SYM s>, <ABSTR s e>, <APPL e1 e2>, <BLEND e1 e2>,
     <VERT e1 e2>, <HORZ e1 e2>, <PROF e1 e2>, <SUSPEND e1 p e2>,
     <TOP e>, <BOT e>, <LEFT e>, <RIGHT e>, <FORE e>, <BACK e>

VAL ::= <p COLOR c>, <p ABSTR s e>, <p BLEND e1 e2>, <p VERT e1 e2>,
     <p HORZ e1 e2>, <p PROF e1 e2>
```

```
ENV ::= NIL, <<s e p>.p'>


==========================================================================================
DESCRIPTION DES ALGORITHMES
==========================================================================================
G-DRAW : VAL x RECT -> PICTURE
G-EVAL : EXP x ENV  -> VAL
G-GET  : SYM x ENV  -> VAL
G-APPLY: VAL x EXP x ENV ->VAL
G-BIND : SYM x EXP x ENV x ENV  -> ENV
G-COLORIZE : COL x VAL -> VAL

------------------------------------------------------------------------------------------
G-DRAW : VAL x RECT -> PICTURE
------------------------------------------------------------------------------------------
G-DRAW(<p VERT e1 e2> r) -> G-DRAW( G-EVAL(e1 p) R-LEFT(r) ) && G-DRAW( G-EVAL(e2 p)
R-RIGHT(r) )
G-DRAW(<p HORZ e1 e2> r) -> G-DRAW( G-EVAL(e1 p) R-TOP(r) ) && G-DRAW( G-EVAL(e2 p)
R-BOTTOM(r) )
G-DRAW(<p PROF e1 e2> r) -> G-DRAW( G-EVAL(e1 p) R-FORE(r) ) && G-DRAW( G-EVAL(e2 p)
R-BACK(r) )
G-DRAW(<p BLEND e1 e2> r) -> G-DRAW(G-EVAL(e1 p) r) &blend& G-DRAW( G-EVAL(e2 p) r )

G-DRAW(<p COLOR c> r)    -> R-PAINT(r c)
G-DRAW(<p ABSTR s e> r)  -> nothing


------------------------------------------------------------------------------------------
G-EVAL : EXP x ENV  -> VAL
------------------------------------------------------------------------------------------
G-EVAL(<IDENT s> p)          -> G-GET(s p)
G-EVAL(<APPL e1 e2> p)       -> G-APPLY( G-EVAL(e1 p) e2 p)
G-EVAL(<SUSPEND e1 p e3> p3) -> G-APPLY( G-EVAL(e1 p) e3 p3)

G-EVAL(<COLOR c> p)          -> <p COLOR c>
G-EVAL(<ABSTR s e> p)        -> <p ABSTR s e>
G-EVAL(<VERT e1 e2> p)       -> <p VERT e1 e2>
G-EVAL(<HORZ e1 e2> p)       -> <p HORZ e1 e2>
G-EVAL(<PROF e1 e2> p)       -> <p PROF e1 e2>
G-EVAL(<BLEND e1 e2> p)      -> <p BLEND e1 e2>

G-EVAL(<TOP e> p)            -> G-TOP(G-EVAL(e p))
G-EVAL(<BOT e> p)            -> G-BOT(G-EVAL(e p))

G-EVAL(<LEFT e> p)           -> G-LEFT(G-EVAL(e p))
G-EVAL(<RIGHT e> p)          -> G-RIGHT(G-EVAL(e p))

G-EVAL(<FORE e> p)           -> G-FORE(G-EVAL(e p))
G-EVAL(<BACK e> p)           -> G-BACK(G-EVAL(e p))



------------------------------------------------------------------------------------------
G-APPLY: VAL x EXP x ENV ->VAL
------------------------------------------------------------------------------------------
G-APPLY(<p ABSTR s e1> e2 p2)  -> G-EVAL(e1 G-BIND(s e2 p2 p))
G-APPLY(<p COLOR c> e2 p)      -> G-COLORIZE(c G-EVAL(e2 p))

G-APPLY(<p VERT e1 e2> e3 p3)  -> <p3 VERT <SUSPEND e1 p e3> <SUSPEND e2 p e3>>
```

```
G-APPLY(<p HORZ e1 e2> e3 p3)  -> <p3 HORZ <SUSPEND e1 p e3> <SUSPEND e2 p e3>>
G-APPLY(<p PROF e1 e2> e3 p3)  -> <p3 PROF <SUSPEND e1 p e3> <SUSPEND e2 p e3>>
G-APPLY(<p BLEND e1 e2> e3 p3) -> <p3 BLEND <SUSPEND e1 p e3> <SUSPEND e2 p e3>>


-------------------------------------------------------------------------------
G-COLORIZE : COL x VAL -> VAL
-------------------------------------------------------------------------------
G-COLORIZE(c <p COLOR c'>)    -> <p COLOR c+c'/2>
G-COLORIZE(c <p ABSTR s e>)   -> <p ABSTR s <APPL <COLOR c> e>>

G-COLORIZE(c <p OPbin e1 e2>) -> <p OPbin <APPL <COLOR c> e1> <APPL <COLOR c> e2>>


-------------------------------------------------------------------------------
G-TOP : VAL -> VAL
-------------------------------------------------------------------------------
G-TOP(<p VERT e1 e2>)  -> <p VERT <TOP e1> <TOP e2>>
G-TOP(<p PROF e1 e2>)  -> <p PROF <TOP e1> <TOP e2>>
G-TOP(<p BLEND e1 e2>) -> <p BLEND <TOP e1> <TOP e2>>
G-TOP(<p COLOR c>)     -> <p COLOR c>
G-TOP(<p ABSTR s e>)   -> <p ABSTR s <TOP e>>

G-TOP(<p HORZ e1 e2>)  -> G-EVAL(e1 p)


-------------------------------------------------------------------------------
G-BOT : VAL -> VAL
-------------------------------------------------------------------------------
G-BOT(<p VERT e1 e2>)  -> <p VERT <BOT e1> <BOT e2>>
G-BOT(<p PROF e1 e2>)  -> <p PROF <BOT e1> <BOT e2>>
G-BOT(<p BLEND e1 e2>) -> <p BLEND <BOT e1> <BOT e2>>
G-BOT(<p COLOR c>)     -> <p COLOR c>
G-BOT(<p ABSTR s e>)   -> <p ABSTR s <BOT e>>

G-BOT(<p HORZ e1 e2>)  -> G-EVAL(e2 p)


-------------------------------------------------------------------------------
G-LEFT : VAL -> VAL
-------------------------------------------------------------------------------
G-LEFT(<p HORZ e1 e2>)  -> <p HORZ <LEFT e1> <LEFT e2>>
G-LEFT(<p PROF e1 e2>)  -> <p PROF <LEFT e1> <LEFT e2>>
G-LEFT(<p BLEND e1 e2>) -> <p BLEND <LEFT e1> <LEFT e2>>
G-LEFT(<p COLOR c>)     -> <p COLOR c>
G-LEFT(<p ABSTR s e>)   -> <p ABSTR s <LEFT e>>

G-LEFT(<p VERT e1 e2>)  -> G-EVAL(e1 p)


-------------------------------------------------------------------------------
G-RIGHT : VAL -> VAL
-------------------------------------------------------------------------------
G-RIGHT(<p VERT e1 e2>)  -> <p VERT <RIGHT e1> <RIGHT e2>>
G-RIGHT(<p PROF e1 e2>)  -> <p PROF <RIGHT e1> <RIGHT e2>>
G-RIGHT(<p BLEND e1 e2>) -> <p BLEND <RIGHT e1> <RIGHT e2>>
G-RIGHT(<p COLOR c>)     -> <p COLOR c>
G-RIGHT(<p ABSTR s e>)   -> <p ABSTR s <RIGHT e>>

G-RIGHT(<p HORZ e1 e2>)  -> G-EVAL(e2 p)
```

```
-----------------------------------------------------------------------------------
G-FORE : VAL -> VAL
-----------------------------------------------------------------------------------
G-FORE(<p VERT e1 e2>)  -> <p VERT <FORE e1> <FORE e2>>
G-FORE(<p HORZ e1 e2>)  -> <p HORZ <FORE e1> <FORE e2>>
G-FORE(<p BLEND e1 e2>) -> <p BLEND <FORE e1> <FORE e2>>
G-FORE(<p COLOR c>)     -> <p COLOR c>
G-FORE(<p ABSTR s e>)   -> <p ABSTR s <FORE e>>

G-FORE(<p PROF e1 e2>)  -> G-EVAL(e1 p)


-----------------------------------------------------------------------------------
G-BACK : VAL -> VAL
-----------------------------------------------------------------------------------
G-BACK(<p VERT e1 e2>)  -> <p VERT <BACK e1> <BACK e2>>
G-BACK(<p HORZ e1 e2>)  -> <p HORZ <BACK e1> <BACK e2>>
G-BACK(<p BLEND e1 e2>) -> <p BLEND <BACK e1> <BACK e2>>
G-BACK(<p COLOR c>)     -> <p COLOR c>
G-BACK(<p ABSTR s e>)   -> <p ABSTR s <BACK e>>

G-BACK(<p PROF e1 e2>)  -> G-EVAL(e2 p)


-----------------------------------------------------------------------------------
G-GET  : SYM x ENV  -> VAL
-----------------------------------------------------------------------------------
G-GET(s NIL)             -> <NIL COLOR black>
G-GET(s <<s e1 p1>.p2>)  -> G-EVAL(e1 p1)
G-GET(s <<s' e1 p1>.p2>) -> G-GET(s p2)


-----------------------------------------------------------------------------------
G-BIND : SYM x EXP x ENV x ENV  -> ENV
-----------------------------------------------------------------------------------
G-BIND (s e p p')  -> <<s e p>.p'>

|#



;/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/
;==================================================================================
;\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
;
;                                        IMPLEMENTATION
;
;/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/
;==================================================================================
;\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\

;==================================================================================
; la compilation des expressions au format interne
;==================================================================================
(defun g-comp-exp (l)
  (cond ((consp l)
         (if (eq (car l) 'color)
           l
           (multiple-value-bind (l1 s l2) (g-split-exp l)
```

```
            (case s
              ($ (list 'PROF (g-comp-exp l1) (g-comp-exp l2)))
              (/ (list 'VERT (g-comp-exp l1) (g-comp-exp l2)))
              (- (list 'HORZ (g-comp-exp l1) (g-comp-exp l2)))
              (+ (list 'BLEND (g-comp-exp l1) (g-comp-exp l2)))
              (= (g-comp-abstr l1 (g-comp-exp l2)))
              ((TOP BOT LEFT RIGHT FORE BACK)
               (if l1
                 (g-comp-appl (cons (list s (g-comp-exp l2)) (reverse l1)))
                 (list s (g-comp-exp l2)) ))
              (otherwise (g-comp-appl (reverse l))) ))))
        (t (list 'IDENT l)) ))

(defun g-split-exp (l2)
  (do (l1)
      ((or (null l2) (member (car l2) '(/ - = + $ TOP BOT LEFT RIGHT FORE BACK)))
(values (nreverse l1) (car l2) (cdr l2)))
    (push (pop l2) l1)))

(defun g-comp-abstr (l v)
  (if l
    (list 'ABSTR (car l) (g-comp-abstr (cdr l) v))
    v))

(defun g-comp-appl (rl)
  (if (cdr rl)
    (list 'APPL (g-comp-appl (cdr rl)) (g-comp-exp (car rl)))
    (g-comp-exp (car rl))))


;==================================================================================
; l'évaluation des expressions
;==================================================================================
(defun g-get (s p)
  (let ((e (assoc s p)))
    (if e
      (g-eval (second e) (third e))
      (list nil 'COLOR nil))))

(defun g-bind (s e p p2)
  (cons (list s e p) p2))

(defun g-eval (e p)
  (case (first e)
    (IDENT (g-get (second e) p))
    (APPL (g-apply (g-eval (second e) p) (third e) p))
    (SUSPEND (g-apply (g-eval (second e) (third e)) (fourth e) p))
    (TOP (g-top (g-eval (second e) p)))
    (BOT (g-bot (g-eval (second e) p)))
    (LEFT (g-left (g-eval (second e) p)))
    (RIGHT (g-right (g-eval (second e) p)))
    (FORE (g-fore (g-eval (second e) p)))
    (BACK (g-back (g-eval (second e) p)))
    (otherwise (cons p e))))

(defun g-apply (v e p)
  (case (second v)
    (ABSTR (g-eval (fourth v) (g-bind (third v) e p (first v))))
    (COLOR (g-colorize (third v) (g-eval e p)))
    (HORZ (list p (second v) (list 'SUSPEND (third v) (first v) (list 'TOP e))
```

```
      (list 'SUSPEND (fourth v) (first v) (list 'BOT e))))
    (VERT (list p (second v) (list 'SUSPEND (third v) (first v) (list 'LEFT e))
      (list 'SUSPEND (fourth v) (first v) (list 'RIGHT e))))
    (PROF (list p (second v) (list 'SUSPEND (third v) (first v) (list 'FORE e))
      (list 'SUSPEND (fourth v) (first v) (list 'BACK e)))) ))

(defun g-colorize (c v)
  (case (second v)
    (COLOR (list nil 'COLOR (mix-colors c (third v))))
    (ABSTR (list (first v) 'ABSTR (third v) (list 'APPL (list 'COLOR c)
      (fourth v))))
    (otherwise (list (first v)
                     (second v)
                     (list 'APPL (list 'COLOR c) (third v))
                     (list 'APPL (list 'COLOR c) (fourth v)))) ))

(defun g-top (v)
  (case (second v)
    (COLOR v)
    (ABSTR (list (first v) 'ABSTR (third v) (list 'TOP (fourth v))))
    (HORZ (g-eval (third v)(first v)))
    (otherwise (list (first v) (second v) (list 'TOP (third v))
      (list 'TOP (fourth v)))) ))

(defun g-bot (v)
  (case (second v)
    (COLOR v)
    (ABSTR (list (first v) 'ABSTR (third v) (list 'BOT (fourth v))))
    (HORZ (g-eval (fourth v)(first v)))
    (otherwise (list (first v) (second v) (list 'BOT (third v))
      (list 'BOT (fourth v)))) ))

(defun g-left (v)
  (case (second v)
    (COLOR v)
    (ABSTR (list (first v) 'ABSTR (third v) (list 'LEFT (fourth v))))
    (VERT (g-eval (third v)(first v)))
    (otherwise (list (first v) (second v) (list 'LEFT (third v))
      (list 'LEFT (fourth v)))) ))

(defun g-right (v)
  (case (second v)
    (COLOR v)
    (ABSTR (list (first v) 'ABSTR (third v) (list 'RIGHT (fourth v))))
    (VERT (g-eval (fourth v)(first v)))
    (otherwise (list (first v) (second v) (list 'RIGHT (third v))
      (list 'RIGHT (fourth v)))) ))

(defun g-fore (v)
  (case (second v)
    (COLOR v)
    (ABSTR (list (first v) 'ABSTR (third v) (list 'FORE (fourth v))))
    (PROF (g-eval (third v)(first v)))
    (otherwise (list (first v) (second v) (list 'FORE (third v))
      (list 'FORE (fourth v)))) ))

(defun g-back (v)
  (case (second v)
    (COLOR v)
    (ABSTR (list (first v) 'ABSTR (third v) (list 'BACK (fourth v))))
    (PROF (g-eval (fourth v)(first v)))
```

```
    (otherwise (list (first v) (second v) (list 'BACK (third v))
      (list 'BACK (fourth v)))) ))

;==================================================================================
; l'interpretation graphique des valeurs
;==================================================================================
(defun g-draw (v w r)
  (unless (apply #'c-empty r)
    (case  (second v)
      (ABSTR nil)
      (COLOR (c-paint w r (third v)))
      (BLEND (g-draw-blend
                   (g-eval (third v) (first v))
                   (g-eval (fourth v) (first v)) w r))
      (PROF (g-draw-prof
                   (g-eval (third v) (first v))
                   (g-eval (fourth v) (first v)) w r))
      (VERT (g-draw-vert
                   (g-eval (third v) (first v))
                   (g-eval (fourth v) (first v)) w r))
      (HORZ (g-draw-horz
                   (g-eval (third v) (first v))
                   (g-eval (fourth v) (first v)) w r)) )))

(defun g-draw-blend (v1 v2 w r)
  (let ((pm (pen-mode w)))
    (g-draw v1 w r)
    (set-pen-mode w 35)
    (g-draw v2 w r)
    (set-pen-mode w pm)))

(defun g-draw-vert (v1 v2 w r)
  (if (apply #'c-small-h r)
    (g-draw (choose-exp 3 v1 v2
                        (not (apply #'c-small-h r))
                        (not (apply #'c-small-v r))
                        (not (apply #'c-small-p r))) w r)
    (progn
      (g-draw v1 w (apply #'c-left r))
      (g-draw v2 w (apply #'c-right r))
      )))

(defun g-draw-horz (v1 v2 w r)
  (if (apply #'c-small-v r)
    (g-draw (choose-exp 3 v1 v2
                        (not (apply #'c-small-h r))
                        (not (apply #'c-small-v r))
                        (not (apply #'c-small-p r))) w r)
    (progn
      (g-draw v2 w (apply #'c-bot r))
      (g-draw v1 w (apply #'c-top r))
      )))

(defun g-draw-prof (v1 v2 w r)
  (if (apply #'c-small-p r)
    (g-draw (choose-exp 3 v1 v2
                        (not (apply #'c-small-h r))
                        (not (apply #'c-small-v r))
                        (not (apply #'c-small-p r))) w r)
    (progn
      (g-draw v2 w (apply #'c-back r))
```

```
      (g-draw v1 w (apply #'c-fore r))
      )))

(defun choose-exp (n v1 v2 lh lv lp)
  (cond ((and lh (eq (second v1) 'VERT)) v1)
        ((and lh (eq (second v2) 'VERT)) v2)
        ((and lv (eq (second v1) 'HORZ)) v1)
        ((and lv (eq (second v2) 'HORZ)) v2)
        ((and lp (eq (second v1) 'PROF)) v1)
        ((and lp (eq (second v2) 'PROF)) v2)
        ((eq (second v1) 'COLOR) v1)
        ((eq (second v2) 'COLOR) v2)
        ((eq (second v1) 'ABSTR) v1)
        ((eq (second v2) 'ABSTR) v2)
        ((> n 0) (choose-exp (- n 1)
                             (choose-exp (- n 1)
                                         (g-eval (third v1) (first v1))
                                         (g-eval (fourth v1) (first v1))
                                         lh lv lp)
                             (choose-exp (- n 1)
                                         (g-eval (third v2) (first v2))
                                         (g-eval (fourth v2) (first v2))
                                         lh lv lp)
                             lh lv lp))
        (t (list nil 'COLOR nil))))

(defun c-calc (kf kb h1 v1 h2 v2)
  (let ((cf (floor (* kf (min (- h2 h1) (- v2 v1)))))
        (cb (floor (* kb (min (- h2 h1) (- v2 v1))))) )
    (list h1 (- v2 cf) (+ h1 cf) v2
          (- h2 cb) v1 h2 (+ v1 cb))))

(defun c-left (da ab bc cd he ef fg gh)
  (list da ab (round (+ da bc) 2) cd   he ef (round (+ he fg ) 2) gh))

(defun c-right (da ab bc cd he ef fg gh)
  (list (round (+ da bc) 2) ab bc cd   (round (+ he fg ) 2) ef fg gh))

(defun c-top (da ab bc cd he ef fg gh)
  (list da ab bc (round (+ ab cd) 2)   he ef fg (round (+ ef gh) 2)))

(defun c-bot (da ab bc cd he ef fg gh)
  (list da (round (+ ab cd) 2) bc cd   he (round (+ ef gh) 2) fg gh))

(defun c-fore (da ab bc cd he ef fg gh)
  (list da ab bc cd
      (round (+ da he) 2)(round (+ ab ef) 2)(round (+ bc fg) 2)
      (round (+ cd gh) 2)))

(defun c-back (da ab bc cd he ef fg gh)
  (list (round (+ da he) 2)(round (+ ab ef) 2)(round (+ bc fg) 2)
      (round (+ cd gh) 2) he ef fg gh))

(defun c-empty (da ab bc cd he ef fg gh)
  (declare (ignore ef fg gh))
  (or (= da bc) (= ab cd) (= da he) ))

(defun c-small-h (da ab bc cd he ef fg gh)
  (declare (ignore ab cd he ef fg gh))
  (>= (+ 1 da) bc))
```

```
(defun c-small-v (da ab bc cd he ef fg gh)
  (declare (ignore da bc he ef fg gh))
  (>= (+ 1 ab) cd))

(defun c-small-p (da ab bc cd he ef fg gh)
  (declare (ignore ab bc cd ef fg gh))
  (>= (+ 1 da) he))

(defun c-paint (w r col)
  (when col
    (incf *g-count*)
    (destructuring-bind (da ab bc cd he ef fg gh) r
      (let (p)

        ; toit
        (start-polygon w)
        (move-to w da ab)
             (line-to w he ef)
             (line-to w fg ef)
             (line-to w bc ab)
             (line-to w da ab)
        (setq p (get-polygon w))
        (set-fore-color w (att-color col 1.2))
        (paint-polygon w p)
        (kill-polygon p)

        ; coté
        (start-polygon w)
        (move-to w bc ab)
             (line-to w fg ef)
             (line-to w fg gh)
             (line-to w bc cd)
             (line-to w bc ab)
        (setq p (get-polygon w))
        (set-fore-color w (att-color col 1.4))
        (paint-polygon w p)
        (kill-polygon p)

        ; face
        (start-polygon w)
        (move-to w da ab)
             (line-to w bc ab)
             (line-to w bc cd)
             (line-to w da cd)
             (line-to w da ab)
        (setq p (get-polygon w))
        (set-fore-color w (att-color col 1))
        (paint-polygon w p)
        (kill-polygon p) ))))

(defun att-color (c n)
  (multiple-value-bind (r g b) (color-values c)
    (make-color  (round r n) (round g n) (round b n))))

(defun mix-colors (c1 c2)
  (if (and c1 c2)
    (multiple-value-bind (r1 g1 b1) (color-values c1)
      (multiple-value-bind (r2 g2 b2) (color-values c2)
        (make-color
             (round (+ (* 3 r1) r2) 4)
             (round (+ (* 3 g1) g2) 4)
```

```
            (round (+ (* 3 b1) b2) 4))))
    (or c1 c2)))

(defun bad-mix-colors (c1 c2)
  (if (and c1 c2)
    (multiple-value-bind (r1 g1 b1) (color-values c1)
      (multiple-value-bind (r2 g2 b2) (color-values c2)
        (make-color  (round (+ r1 r2) 2) (round (+ g1 g2) 4) (round (+ b1 b2) 4))))
    (or c1 c2)))

;==================================================================================
; la fenetre d'affichage graphique
;==================================================================================
(defclass t-view (window)
  ((txt :initform nil)
   (exp :initform nil)
   (prof :initform 64)
   (kf :initform 0.9)
   (kb :initform 0.7)
   ))

;; méthodes redefinies

(defmethod pen-mode ((view t-view))
  (rref (wptr view) windowRecord.pnmode))

(defmethod set-pen-mode ((view t-view) new-mode)
  (with-port (wptr view) (#_PenMode new-mode)))

(defmethod window-close ((self t-view))
  (window-hide self))

(defmethod set-view-size ((self t-view) h &optional v)
  (declare (ignore h v))
  (without-interrupts
   (invalidate-view self t)
   (call-next-method)))

(defmethod view-draw-contents ((self t-view))
  (let ((h1 (point-h (view-scroll-position self)))
        (h2 (point-h (add-points (view-scroll-position self)(view-size self))))
        (v2 (point-v (add-points (view-scroll-position self)(view-size self))))
        (v1 (+ 16 (point-v (view-scroll-position self)))))

    (set-fore-color self *black-color*)
    (set-pen-mode self 0)
    (paint-rect self (- h1 4) (- v1 4) (+ h2 4) (+ v2 4))

    (move-to self (+ h1 2) (- v1 6)) (format self "~S" (my txt))
    (g-draw (my exp) self
      (c-calc (my kf) (my kb) (+ 8 h1) (+ 4 v1) (- h2 8) (- v2 8)))
    ))

(defun display (e)
  (setq *g-count* 0)
  (unless *g-wind* (setq *g-wind* (new 't-view
                                       :window-title "g-calcul"
                                       :color-p t
                                       :view-size #@(203 182))))
  (let ((self *g-wind*))
    (my txt e)
```

```
    (my exp (g-eval (g-comp-exp e) *g-env*))
    (invalidate-view self t)
    (unless (window-shown-p self) (window-show self))
    (set-window-layer self 0))
  *g-count*)


;==================================================================================
; outils pour l'utilisation
;==================================================================================

(defun df (nom exp)
  (setq *g-env* (g-bind nom (g-comp-exp exp) *g-env* *g-env*))
  nom)

(defun g-bootstrap ()
  (setq *g-env* nil)

  (df 'invisible (list 'color nil))
  (df 'red (list 'color (make-color 65535 0 0)))
  (df 'green (list 'color (make-color 0 65535 0)))
  (df 'blue (list 'color (make-color 0 0 65535)))
  (df 'white (list 'color (make-color 65535 65535 65535)))
  (df 'black (list 'color 0))

  (df 'true '(x y = x))
  (df 'false '(x y = y))
  (df 'or '(x y = x true y))
  (df 'and '(x y = x y false))
  (df 'not '(x = x false true))
  (df 'cons '(a b s = s a b))
  (df 'first '(c = c true))
  (df 'rest '(c = c false))
  (df 'y '(f = (x = f (x x)) (x = f (x x))))
  (df 'hole '(y (f x = f)))
  (df 'id '(x = x))
  (df 'linf '(y (inf op c1 c2 = op c1 (inf op c2 c1))))
  (df 'finf '(y (inf op c1 c2 = op (inf op c2 c1) c1)))
  (df 'pv '(c1 c2 = c1 - c2))
  (df 'ph '(c1 c2 = c1 / c2))
  (df 'pp '(c1 c2 = c1 $ c2))
  (df 'aleft '(linf ph))
  (df 'aright '(finf ph))
  (df 'atop '(finf pv))
  (df 'abot '(linf pv))
  (df 'rot '(y (f c1 c2 = (c1 / f c2 c1) - (f c2 c1 / c2))))
  (df 'bl '(c = (bot (left c))))
  (df 'br '(c = (bot (right c))))
  (df 'tl '(c = (top (left c))))
  (df 'tr '(c = (top (right c))))
  (df 'rclock '((bl / tl) - (br / tr)))
  (df 'rp '(y (f c = f (back c) $ (fore c))))
  (df 'truc '(c1 c2 = (c1 / c2) - c1))
  )

(g-bootstrap)

(defmacro disp (exp)
  `(display ',exp))

(defmacro new-disp (exp)
```

```
  `(progn (setq *g-wind* nil) (display ',exp)))

(defmacro disp1 (exp)
  `(progn (setq *g-wind* *g-w1*) (display ',exp) (setq *g-w1* *g-wind*)))

(defmacro disp2 (exp)
  `(progn (setq *g-wind* *g-w2*) (display ',exp) (setq *g-w2* *g-wind*)))

(defmacro disp3 (exp)
  `(progn (setq *g-wind* *g-w3*) (display ',exp) (setq *g-w3* *g-wind*)))

(defmacro disp4 (exp)
  `(progn (setq *g-wind* *g-w4*) (display ',exp) (setq *g-w4* *g-wind*)))


;==================================================================================
; exemple d'utilisation
;==================================================================================

#|

(disp (aright (atop green red) (abot green red)))
(disp ((aleft - aright) ((abot / atop) red green) ((abot / atop) green blue)))
(disp ((abot / atop) green blue))
(disp (rot abot atop green ((aleft - aright) ((abot / atop) red green) ((abot /
atop) green blue))))
(disp (abot aright aleft aleft (atop / abot) blue red))
(disp (y (f c1 c2 = (c1 / f c2 c1) - (f c1 c2 / c1)) blue red))

(disp (top atop red blue))
(disp (top (y (f c1 c2 = (c1 / f c2 c1) - (f c1 c2 / c1)) blue red)))

(disp (rclock ((red / blue) - (green / red))))
(disp (rclock ((red - green) / (blue - red))))

(disp (y (f c1 c2 = (c1 / f c2 c1) - (f c1 c2 / c1)) blue red))
(disp (rclock (y (f c1 c2 = (c1 / f c2 c1) - (f c1 c2 / c1)) blue red)))

(disp ((rclock (c1 c2 = (c1 / c2) - (c2 / (c1 / c2)))) blue red))
(disp ((rclock (c1 c2 = (c1 / c2) - (c2 / (c1 / c2)))) atop aleft blue red))

(disp (rot abot atop invisible ((aleft - aright) ((abot / atop) red invisible)
((abot / atop) invisible blue))))

(disp ((rot abot atop invisible ((aleft - aright) ((abot / atop) red invisible)
((abot / atop) invisible blue))) + white))

(disp ((c1 c2 = ((invisible / (invisible $ c2)) - ((c1 $ invisible) / invisible)))
red red))

(disp (linf (c1 c2 = ((invisible / (invisible $ c2)) - ((c1 $ invisible) /
invisible))) red red))

(disp (linf (c1 c2 = ((invisible / (invisible $  c2 + white)) - ((c1 $ invisible) /
invisible))) red red))

(disp (rp (linf (c1 c2 = ((invisible / (invisible $  c2)) - ((c1 $ invisible) /
invisible))) red blue)))

(disp ( ((id / ( id - ((id - (id / hole)) / id))) - id) (linf (c1 c2 = ((hole /
(hole $  c2  c1)) - ((c1 $ hole) / hole))) red blue)))
```

```
(disp ( (truc (truc id (truc  hole id)) id) (linf (c1 c2 = ((hole / (hole $  c2
c1)) - ((c1 $ hole) / hole))) red blue)))


(progn
  (g-bootstrap )
  ;(df 'gru '(c1 c2 = (c1 / c2) - (c2 / c1)))
  (df 'gru '((hole / id) - (id / hole)))
  (df 'split '(c = (c / c) - (c / c)))
  (disp ( (split (split (split (split gru)))) (linf (c1 c2 = ((hole / (hole $  c2
c1)) - ((c1 $ hole) / hole))) red blue))))

(progn
  (g-bootstrap )
  ;(df 'gru '(c1 c2 = (c1 / c2) - (c2 / c1)))
  (df 'gru '((hole / id) - (id / hole)))
  (df 'grup '((hole / id) $ (id / hole)))
  (df 'split '(c = (c / c) - (c / c)))
  (df 'splitp '(c = (c / c) $ (c / c)))
  (disp ( (splitp (splitp (splitp (splitp grup))))(split (split (split (split
gru))))  (linf (c1 c2 = ((hole / (hole $  c2  c1)) - ((c1 $ hole) / hole))) red
blue))))

(progn
  (g-bootstrap )
  (df 'gru '((hole / id) - (id / hole)))
  (df 'split '(c = (c / c) - (c / c)))
  (disp ( (finf pp (split (split (split gru))) (split (split (split (split gru)))))
(linf (c1 c2 = ((hole / (hole $  c2  c1)) - ((c1 $ hole) / hole))) red blue))))

(progn
  (g-bootstrap )
  (df 'tri1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'tri2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'losange '(c1 c2 = (tri1 c1 c2 / tri2 c1 c2) - (tri2 c2 c1 / tri1 c2 c1)))
  (disp (losange red blue)))

(progn
  (g-bootstrap )
  (df 'tri1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'tri2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'losange '(c1 c2 = (tri1 c1 c2 / tri2 c1 c2) - (tri2 c2 c1 / tri1 c2 c1)))
  (disp (losange (losange green blue) red))) ;; plante

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangef '(c1 c2 = (trif1 c1 c2 / trif2 c1 c2) - (trif2 c2 c1 / trif1 c2
c1)))
  (df 'losangep '(c1 c2 = (trip1 c1 c2 $ trip2 c1 c2) - (trip2 c2 c1 $ trip1 c2
c1)))
  (disp ((losangef (or false) (or true)) (losangep true false) hole red)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
```

```
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangef '(c1 c2 = (trif1 c1 c2 / trif2 c1 c2) - (trif2 c2 c1 / trif1 c2
c1)))
  (df 'losangep '(c1 c2 = (trip1 c1 c2 $ trip2 c1 c2) - (trip2 c2 c1 $ trip1 c2
c1)))
  (disp (losangep blue red)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangef '(c1 c2 = (trif1 c1 c2 / trif2 c1 c2) - (trif2 c2 c1 / trif1 c2
c1)))
  (df 'losangep '(c1 c2 = (trip1 c1 c2 $ trip2 c1 c2) - (trip2 c2 c1 $ trip1 c2
c1)))
  (disp1 (losangef blue red))
  (disp2 (losangep blue red))
  (disp3 (and (losangef false true) (losangep true false) blue red)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangef '(c1 c2 = (trif1 c1 c2 / trif2 c1 c2) - (trif2 c2 c1 / trif1 c2
c1)))
  (df 'losangep '(c1 c2 = (trip1 c1 c2 $ trip2 c1 c2) - (trip2 c2 c1 $ trip1 c2
c1)))
  (disp (and (losangef false true) (losangep true false) hole red)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangef '(c1 c2 = (trif1 c1 c2 / trif2 c1 c2) - (trif2 c2 c1 / trif1 c2
c1)))
  (df 'losangep '(c1 c2 = (trip1 c1 c2 $ trip2 c1 c2) - (trip2 c2 c1 $ trip1 c2
c1)))
  (disp (and (losangef false true) (losangep true false)  red hole)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangef '(c1 c2 = (trif1 c1 c2 / trif2 c1 c2) - (trif2 c2 c1 / trif1 c2
c1)))
  (df 'losangep '(c1 c2 = (trip1 c1 c2 $ trip2 c1 c2) - (trip2 c2 c1 $ trip1 c2
c1)))
  (disp (not (and (losangef false true) (losangep true false)) hole red )))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
```

```
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangef '(c1 c2 = (trif1 c1 c2 / trif2 c1 c2) - (trif2 c2 c1 / trif1 c2
c1)))
  (df 'losangep '(c1 c2 = (trip1 c1 c2 $ trip2 c1 c2) - (trip2 c2 c1 $ trip1 c2
c1)))
  (disp ( (losangef (and false) (and true)) (losangep false true) blue hole)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (disp ( (trif1 red blue) / (trif2 red blue))))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (disp ( (trif1 / trif2) red blue)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (disp (((trif1 / trif2) - (trif2 / trif1)) red blue)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'losangef '(c1 c2 = (((trif1 / trif2) c2 c1) - ((trif2 / trif1) c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangep '(c1 c2 = (((trip1 $ trip2) c2 c1) - ((trip2 $ trip1) c1 c2))))
  (disp (losangep (losangep red hole) hole)))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'losangef '(c1 c2 = (((trif1 / trif2) c2 c1) - ((trif2 / trif1) c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangep '(c1 c2 = (((trip1 $ trip2) c2 c1) - ((trip2 $ trip1) c1 c2))))
  (disp ((finf pp id hole) (losangef (losangef red (red white)) (losangef blue (blue
white))))))

(progn
  (g-bootstrap )
  (df 'trif1 '(y (f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2))))
  (df 'trif2 '(y (f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2))))
  (df 'losangef '(c1 c2 = (((trif1 / trif2) c2 c1) - ((trif2 / trif1) c1 c2))))
  (df 'trip1 '(y (f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2))))
  (df 'trip2 '(y (f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2))))
  (df 'losangep '(c1 c2 = (((trip1 $ trip2) c2 c1) - ((trip2 $ trip1) c1 c2))))
  (disp ((trif2 hole id) (trip2 id hole) red)))


(progn
  (g-bootstrap )
```

```
  (df 'trif1 '(f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2)))
  (df 'trif2 '(f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2)))
  (df 'losangef '(c1 c2 = (((trif1 / trif2) c2 c1) - ((trif2 / trif1) c1 c2))))
  (df 'trip1 '(f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2)))
  (df 'trip2 '(f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2)))
  (df 'losangep '(c1 c2 = (((trip1 $ trip2) c2 c1) - ((trip2 $ trip1) c1 c2))))
  (disp (y (trip1 / trip2) (red - hole) hole)))


(progn
  (g-bootstrap )
  (df 'trif1 '(f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2)))
  (df 'trif2 '(f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2)))
  (df 'losangef '(c1 c2 = (((trif1 / trif2) c2 c1) - ((trif2 / trif1) c1 c2))))
  (df 'trip1 '(f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2)))
  (df 'trip2 '(f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2)))
  (df 'losangep '(c1 c2 = (((trip1 $ trip2) c2 c1) - ((trip2 $ trip1) c1 c2))))
  (disp (y trip1 hole red (y (y trif1 trip1 trip2) red hole))))


(progn
  (g-bootstrap )
  (df 'trif1 '(f c1 c2 = (c1 / f c1 c2) - (f c1 c2 / c2)))
  (df 'trif2 '(f c1 c2 = (f c1 c2 / c1) - ( c2 / f c1 c2)))
  (df 'losangef '(c1 c2 = (((trif1 / trif2) c2 c1) - ((trif2 / trif1) c1 c2))))
  (df 'trip1 '(f c1 c2 = (c1 $ f c1 c2) - (f c1 c2 $ c2)))
  (df 'trip2 '(f c1 c2 = (f c1 c2 $ c1) - ( c2 $ f c1 c2)))
  (df 'losangep '(c1 c2 = (((trip1 $ trip2) c2 c1) - ((trip2 $ trip1) c1 c2))))
  (disp (y (y trif2 (y trif1 trip1 trip2) (y trif1 trip1 trip2)) hole red)))


    |#
```

7. BIBLIOGRAPHIE

[Abelson, Sussman 1989] - Structure et interprétation des programmes informatiques InterEditions 1989.

[Andréani 1986] - L'écriture montage - Conséquences n°8

[Arya 1989] - Processes in a Functional Animation System - FPCA '89, The Fourth International Conference on Functional Programming Languages and Computer Architecture. ACM Press, New York, NY.

[Backus 1978] - Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs - Communications of the ACM 21,8.

[Baisnée, Barrière , Dalbavie, Duthen, Lindberg, Poland, Saariaho 1988] - Esquisse : a compositionnal environment. - Proceedings of the International Computer Music Conference 1988, Computer Music Association, San Francisco.

[Barriere 1988] - L'informatique musicale comme approche cognitive : simulation, timbre et processus formels. dans 'La musique et les sciences cognitives' - S. McAdams / I. Deliège ed. Pierre Mardaga.

[Barriere 1990] - Devenir de l'écriture musicale assistée par ordinateur : formalisme, forme, aide à la composition. - Analyse musicale 20.

[Berry, Couronné, Gonthier 1987] - Programmation synchrone des systèmes réactifs : le langage ESTEREL- Techniques et Sciences Informatiques Vol 6 N°4 1987.

[Bestougeff, Ligozat 1989] - Outils logiques pour le traitement du temps - Masson, Paris.

[Boynton, Lavoie, Orlarey, Rueda, Wessel 1986] - MIDI-LISP, a LISP-based music programming environment for the Macintosh - Proceedings of the International Computer Music Conference 1986, Computer Music Association, San Francisco, CA.

[Burstall, MacQueen, Sanella 1980] - HOPE : An experimental applicative language - Proceeding LISP Conference, Stanford, CA.

[Chailloux, Devin, Dupont, Hullot, Serpette, Vuillemin 1986] - Le_Lisp de l'INRIA version 15.2, Le manuel de référence - INRIA, Rocquencourt.

[Church 1941] - The Calculi of lambda-conversion. Princeton University Press.

[Cointe, Rodet 1983] - Formes: a new object-language for managing a hierarchy of events - IRCAM, Paris.

[Curry 1951] - La théorie des combinateurs; la logique combinatoire et les antinomies - Rendiconti di matematica e delle sue applicazioni. Societa Italiana di Logica e Filosofia delle Scienze.

[Dannenberg 1984] - Artic: A Functional Language for Real-Time Control - ACM Symposium on LISP and Functional Programming, ACM.

[Desain P. Honing H. 1991] - Tempo curves considered harmful - Proceedings of the International Computer Music Conference 1991, Computer Music Association, San Francisco, CA.

[Desclés 1990] - Langages applicatifs langues naturelles et cognition - Hermès, Paris.

[Dijkstra 1972] - The Humble Programmer ACM Turing Award Lectures 1972

[Dufour 1981] - L'artifice d'écriture. - Paris, Critique n°408.

[Ferneyhough 1986] - La notation et la composition : aspects. - Conséquences n°8

[Field, Harrison 1988] - Functionnal programming - Addison-Wesley Publishers.

[Fober, Orlarey 1992] - Notation et représentation des processus compositionnels dans les langages d'assistance à la composition musicale - Rapport de recherche - Grame, Lyon.

[Goldfarb Charles F.] - The SGML handbook - Oxford University Press, 1990.

[Halbwachs,Caspi,Raymond,Pilaud 1991] - Programmation et vérification des systèmes réactifs : le langage LUSTRE- Techniques et Sciences Informatiques Vol 10 N°2 1991

[Hudak 1989] - Conception, Evolution, and Application of Functional Programming Languages- ACM Computer Surveys Vol 21 N°3 Septembre 1989.

[Huges 1989] - Why Functional Programming Matters- The Computer Journal Vol 32 N°2 1989.

[Iverson 1979] - Notation as a Tool of Thought- in ACM Turing Award Lectures, ACM Press Anthology Series, Addison-Wesley, 1987.

[Jones, Sinclair 1989] - Functional Programming and Operating Systems- The Computer Journal Vol 32 N°2 1989.

[Kelly 1989] - Functional Programming for Loosely-coupled Multiprocessors - MIT Press, Cambridge, Mass.

[Knight 1989] - Unification: a multidisiplinary survey- ACM computing surveys Vol 21 N° 1 pp 93-124 1989.

[Krivine 1990] - Lambda-calcul, types et modèles, Masson, Paris.

[Kurkela 1988] - Partition, vision, action. dans 'La musique et les sciences cognitives" - S. McAdams / I. Deliège ed. Pierre Mardaga.

[Laurson, Duthen 1989] - PATCHWORK a Graphic Language in preFORM - Proceedings of the International Computer Music Conference 1989, Computer Music Association, San Francisco.

[Lequay 1986] - Manuel d'utilisation MidiLogo - Grame, Lyon.

[Letz, Merlier, Orlarey 1990] - Common Lisp Compositional Environment : Manuel de Référence - Grame, Lyon.

[Lonchamp 1989] - Les langages de programmation- Masson, Paris.

[MacLennan 1990] - Functional Programming, practice and theory - Addison-Wesley, Reading, Mass.

[Massini, Napoli, Colnet, Leonard, Tombre 1989] - Les langages à objets- InterEditions Paris 1989

[Mathews 1961] - An Acoustical compiler for music and psychological stimuli- Bell System Technical Journal 40 1961.

[Mathews 1969] -The technology of computer music - MIT press, Cambridge, Mass.

[Meeùs 1991] - Apologie de la partition - Analyse Musicale 24.

[Newcomb Steven R. , Kipp Neill A., Newcomb Victoria T. - Oct. 1991] - The "HyTime" Hypermedia / Time-based Document Structuring Language - SGML SIGhyper Newsletter, October 1191.

[Nicolas 1991] - Huit thèses sur l'écriture musicale - Analyse musicale 24.

[Oppenheim 1989] - DMIX : An environment for composition - Proceedings of the International Computer Music Conference 1989, Computer Music Association, San Francisco.

[Orlarey 1984] - M-LOGO : un langage de programmation orienté composition musicale - Proceedings of the International Computer Music Conference 1984, Computer Music Association, San Francisco.

[Orlarey 1986] - Manuel de référence MidiLogo - Grame, Lyon.

[Orlarey 1986] - MidiLogo : a Midi Composing Environment for the Apple IIe - Proceedings of the International Computer Music Conference 1986, Computer Music Association, San Francisco.

[Orlarey 1990] - An Efficient Scheduling Algorithm for Real-Time Musical Systems - Proceedings of the International Computer Music Conference 1990, Computer Music Association, San Francisco.

[Orlarey, Lequay 1989] - MidiShare : a Real Time multi-tasks software module for Midi applications - Proceedings of the International Computer Music Conference 1989, Computer Music Association, San Francisco.

[Orlarey, Lequay 1989] - MidiShare : manuel de programmation - Grame, Lyon.

[Papert 1980] - Mindstorms : children, computers, and powerful ideas - Basic Books, New York 1980.

[Peyton Jones 1990] - Mise en oeuvre des langages fonctionnels de programmation - Masson, Paris.

[Pingali 1990] - Lazy Evaluation and the Logic variable - In Research Topics in Functional Programming Ed D . Turner Addison-Wesley Publishing Compagny 1990.

[Pinson, Wiener 1991] - Objective-C : Object-Oreinted Programming Techiques - Addison-Wesley Publishers.

[Puckette 1988] - The Patcher - Proceedings of the International Computer Music Conference 1988, Computer Music Association, San Francisco.

[Rich Robert P. 1975] - Musical Notation for Computer Input - Bulletin of American Society for Information Science, Vol. 2, n° 5.

[Shapiro 1989] - The Family of Concurent Logic Programming Languages - ACM Computer Surveys Vol 21 N°3 Septembre 1989.

[Smoliar S.W. 1972] - Music Theory - A Programming Linguistic Approach. Assoc. Computing Machinery, 25th annual conference.

[Schottstaedt 1983] - Pla : a composer's idea of a language - Computer Music Language, spring 1983.

[Turner 1986] - An overview of Miranda : ACM SIGPLAN Notices 21(12) décembre 1986.

[Warnock John E. - 1992] - The new age of documents - Byte, June 1992.

[Wright Haviland - 1992] - SGML frees information - Byte, June 1992.