transformer leurs entrées : une macro LISP *est* un programme LISP. Comparez ceci au pré-processeur du C par exemple, qui ne sait faire que de la transformation lexicale (une macro CPP ne peut pas exécuter de code C pour calculer son résultat). Comparez également ceci aux « templates » de C++ dont la syntaxe est notoirement lourde.

Quand vous écrivez une application LISP, vous pouvez donc considérer que votre code est composé d'expressions LISP qui seront évaluées à des moments différents : au parsing du code source (d'où extension de la syntaxe possible), à la compilation (d'où code statique), et finalement à l'exécution de votre programme.

Les macros LISP (de surcroît combinées à la réflexivité du langage) sont la raison pour laquelle on dit que LISP est un langage de programmation programmable : on peut autant adapter (entendre « modifier ») le langage à ses besoins qu'on est obligé d'adapter ses besoins au langage.

Il existe de nombreux autres aspects libertaires dans LISP (par exemple la possibilité de mélanger les formes de scoping). Mais ce qu'il faut surtout retenir, c'est que LISP est un langage multi-paradigme : si vous voulez être fonctionnel pur, vous pouvez mais ce n'est pas une obligation. Si vous voulez être impératif, c'est pareil. Si vous voulez être orienté objet, c'est la même chose. Et tous les paradigmes peuvent être utilisés (ou pas) à loisir au sein de la même application. Du point de vue de l'expressivité, LISP est véritablement le langage de la liberté !

En bref

Les nouvelles publications (disponibles sur publis.lrde.epita.fr)

- **BAILLARD, A., BERTIN, E., MELLIER, Y., MCCRACKEN, H. J., GÉRAUD, T., PELLÓ, R., LEBORGNE, J.-F., AND FOUQUÉ, P.**. Project EFIGI : Automatic classification of galaxies. In Gabriel, C., Arviset, C., Ponz, D., and Solano, E., editors, *Astronomical Data Analysis Software and Systems XV*, volume 351 of *Conference*, pages 236–239. Astronomical Society of the Pacific
- **CADILHAC, M., HÉRAULT, T., LASSAIGNE, R., PEYRONNET, S., AND TIXEUIL, S.**. Evaluating complex MAC protocols for sensor networks with APMC. In *Proceedings of the 6th International Workshop on Automated Verification of Critical Systems (AVoCS)*, Electronic Notes in Theoretical Computer Science Series
- **CHEKROUN, M., DARBON, J., AND CIRIL, I.**. On a polynomial vector field model for shape representation. In *Proceedings of the International Conference onImage Analysis and Recognition (ICIAR)*, Povoa de Varzim, Portugal. Springer-Verlag
- **DARBON, J., LASSAIGNE, R., AND PEYRONNET, S.**. Approximate probabilistic model checking for programs. In *Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06)*, Technical University of Cluj-Napoca, Romania
- **DAVID, V., DEMAILLE, A., AND GOURNET, O.**. Attribute grammars for modular disambiguation. In *Proceedings of the IEEE 2nd International Conference on Intelligent Computer Communication and Processing (ICCP'06)*, Technical University of Cluj-Napoca, Romania
- **DEMAILLE, A., PEYRONNET, S., AND SIGOURE, B.**. Modeling of sensor networks using XRM. In *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'06)*, Coral Beach Resort, Paphos, Cyprus
- **DENISE, A., GAUDEL, M.-C., GOURAUD, S.-D., LASSAIGNE, R., AND PEYRONNET, S.**. Uniform random sampling of traces in very large models. In *Proceedings of Random Testing 2006 (RT06)*, ACM digital library
- **HÉRAULT, T., LASSAIGNE, R., AND PEYRONNET, S.**. APMC 3.0 : Approximate verification of discrete and continuous time markov chains. In *Proceedings of Qest 2006*
- **LESAGE, D., DARBON, J., AND AKGUL, C.**. An efficient algorithm for connected attribute thinnings and thickenings. In *Proceedings of the second International Conference on Visual Computing*, Lecture Notes in Computer Science Series, Lake Tahoe, Nevada, USA. Springer-Verlag
- **VERNA, D.**. Beating C in scientific computing applications. In *Third European LISP Workshop at ECOOP*, Nantes, France. **Best Paper Award**
- **VERNA, D.**. How to make LISP go faster than C. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, Hong Kong. International Association of Engineers. **Best Paper Award**
- **VERNA, D.**. Latex curricula vitae with the CurVe class. *The PracTEX Journal*, 2006(3)

Les logiciels Vaucanson 1.0 est sorti le 28 juillet 2006. CurVe 1.11[7], une classe de Curriculum Vitae pour LaTeX2e, a vu le jour le 7 juin 2006.

[7] CurVe 1.11, http://www.lrde.epita.fr/~didier/comp/development/software.php#curve.

L'aléastriel du **L**aboratoire de **R**echerche et de **D**éveloppement de l'**E**PITA[1] Numéro 6, Septembre 2006

L'air de rien N° 6

Édito

par *Akim Demaille (Enseignant-Chercheur)*

Il n'est pas d'école d'ingénieurs sans un laboratoire de recherche actif, qui contribue à la renommée de l'école par ses publications et ses collaborations avec d'autres institutions, qui pousse l'enseignement des fondements théoriques mais aussi de l'état de l'art, et qui tisse un lien avec l'industrie au travers de contrats de recherche. C'est le rôle que le LRDE veut jouer au sein de l'EPITA.

L'air de rien, le bulletin du LRDE, vous informera tout au long de l'année des actualités du labo en recherche, en développement, et en enseignement. Ce premier numéro de l'année scolaire 2006/7 vous présente nos enseignants et leurs enseignements du premier term, vous introduit la publication récente du projet VAUCANSON, vous en apprend plus sur la vraie nature du Lisp, et, comme toujours, vous rapporte brièvement les événements qui marquent la vie du labo. Pour en savoir plus, consultez notre site[2], et en particulier nos actualités[3].

Les enseignements

Les membres du LRDE dispensent des cours[4] tant en tronc commun, qu'en options.

THL — THÉORIE DES LANGAGES **A. Demaille**

Akim Demaille est ingénieur et docteur en informatique de l'ENST. Il s'intéresse à la théorie des langages de programmation, la construction des compilateurs et la transformation de programmes. Il encadre les projets Vaucanson et Transformers (*front-end* générique pour le C++).

Ce cours de 28h vise à définir rigoureusement les « langages », à se donner les moyens de spécifier ces objets souvent infinis, et à montrer comment on peut dériver de leur définition des algorithmes d'analyse. Ces techniques sont utilisées par les compilateurs, interpréteurs, navigateurs, et autres outils à format de données structuré (par exemple XML, LaTeX, Dot, etc.).

SEXP — SYSTÈMES D'EXPLOITATION **D. Verna**

Didier Verna est ingénieur et docteur en informatique de l'École Nationale Supérieure des Télécommunications. Il s'intéresse aux langages fonctionnels (LISP en particulier) et au mélange de paradigmes au sein de ceux-ci (orientation objet, métaprogrammation *etc.*). Il est également intéressé par la synthèse d'images et la typographie. Il enseigne aussi à l'ENST, l'ENSTA et au Mastère d'Informatique de Jussieu.

Ce cours de 28h a pour but de fournir aux étudiants une culture générale sur les systèmes d'exploitation, ainsi que la connaissance fondamentale relative à leur fonctionnement. On étudiera l'ensemble des problématiques communes à tous les systèmes (gestion des processus, de la mémoire *etc.*), ainsi que les différentes solutions qui peuvent y être apportées.

THJX — THÉORIE DES JEUX **S. Hémon**

Sébastien Hémon est doctorant au LRI (Laboratoire de Recherche en Informatique) situé à Orsay. Diplômé d'un DEA en logique et ancien professeur de mathématiques du secondaire, il s'intéresse surtout aux aspects théoriques de l'informatique. La théorie des jeux est son domaine de prédilection et le LRDE lui fournit une vision plus appliquée et pratique de ses travaux qui portent principalement sur le calcul effectif des équilibres de Nash et la recherche

[1] L'air de rien, http://publis.lrde.epita.fr/LrdeBulletin.
[2] Site du LRDE, http://www.lrde.epita.fr.
[3] Actualités du LRDE, http://www.lrde.epita.fr/Lrde/News.
[4] Cours assurés par le LRDE, http://epita.lrde.epita.fr/CourseList.

de stratégies gagnantes. Il passe également du temps à travailler sur une branche très particulière : le calcul quantique.

Ce cours de 14h découvre de façon attractive la théorie des jeux dans sa globalité. En passant par les différentes représentations de jeux possibles et leurs classifications principales, l'objectif est de connaître les différents aspects d'un jeu que l'on peut étudier suivant une représentation. On modélisera quelques jeux afin de résoudre des problèmes spécifiques et on verra comment tirer profit de l'algorithmique.

Qu'est-ce que la théorie des jeux ? Si à l'origine cette branche des mathématiques s'intéressait à l'étude de jeux au moyen de modèles formels, elle est devenue un pôle d'attraction où se rencontrent mathématiciens, informaticiens, économistes, biologistes et même physiciens. Ceci parce que les modèles théoriques des jeux offrent d'autres interprétations que les applications ludiques originelles. Ainsi, certains jeux de pions sont isomorphes (lire « équivalents ») à des problèmes de factorisation d'entiers, des jeux de pièces à plusieurs servent de base à des protocoles de sécurité informatique. Si l'étude d'un jeu peut s'avérer amusante, à partir d'un certain moment il faut une grande rigueur pour pouvoir comprendre en détails les mécanismes profonds. Les jeux vont devenir mécanismes, de sorte que l'on aurait pu appeler cela *théorie des processus structurés de décision*.

VAUCANSON 1.0

Vous reprendrez bien un peu d'automate ?
par *Michaël Cadilhac* (EPITA *2007, Option* CSI)

Parmi les événements qui tiennent à cœur aux gens du LRDE, un des plus importants reste la sortie d'une version d'un de nos projets. Après le passage du compilateur TIGER en version 1.0, le projet VAUCANSON [5], la plate-forme de manipulation d'automates finis, n'allait pas être longtemps en reste.

La genèse de ce projet se situe à l'ENST lors du stage de deux étudiants du laboratoire en 2002 sous la houlette de Jacques Sakarovitch, Directeur de Recherche du CNRS, et de Sylvain Lombardy, maintenant professeur à l'université de Marne-la-Vallée. Puis, telle une course de relais des temps modernes, la construction et la maintenance de VAUCANSON passaient aux promotions CSI suivantes, toujours guidés par Jacques et Sylvain.

Le LRDE est aujourd'hui fier de prétendre que VAUCANSON a atteint un point de non-retour dans son utilisabilité, en arborant ostentatoirement son premier nombre de révision majeure non nul.

La fin... Posons le contexte : nous voulons des automates, ces objets mathématiques que l'on représente sous forme de graphe et dont on suit « l'exécution » en considérant un mot d'entrée et en parcourant les transitions étiquetées par des lettres. Distinguant certains états dits *finals*, on dira qu'un mot est *accepté* par l'automate si ce procédé aboutit à un de ceux là.

Mais ces machines à états finis, si elles n'étaient que ça, n'auraient pas plus d'expressivité que `grep(1)` ! La théorie des automates décrit donc des automates qui ne disent pas seulement si un mot a été *reconnu* ou non par l'automate, mais *évaluent* le mot d'entrée, en renvoyant un nombre, un réel et pourquoi pas un autre mot.

C'est ainsi que l'on peut créer des automates qui comptent le nombre de *a* dans un mot, qui divisent un nombre en représentation binaire par 3, qui donnent la distance de Levenshtein entre deux mots, ou qui implémentent `rot13`.

Un des buts de VAUCANSON est donc de pouvoir gérer tous ces types d'automates sans duplication de code, dans un environnement souple et puissant, où les spécialistes du domaine peuvent reconnaître leurs outils mathématiques et où les néophytes peuvent découvrir aisément l'interactivité et la force de ces outils.

Avec un éventail de plus de 50 algorithmes sur automates et fort d'un important couplage avec les réalités mathématiques, VAUCANSON est arrivé à un stade de maturité fonctionnelle non négligeable. Il est d'ailleurs question de s'en servir en TP pour les classes prépas intégrées de l'EPITA.

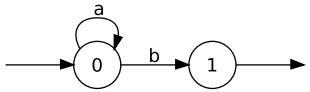
...et les moyens ! Visant généricité et performance, le développement en C++ de VAUCANSON est passé par la création d'un *design pattern*, dont l'expressivité se rapproche des structures mathématiques en jeu. L'expérience des gens d'OLENA [6] a permis d'apporter une expertise toute particulière à cette création.

VAUCANSON est de nos jours un groupement de plusieurs modules : la bibliothèque — au sens *headers* de la chose —, les outils d'entrée et de sortie — le projet définissant un format XML pour la représentation des automates —, et les outils en ligne de commande implémentés à l'aide de la bibliothèque au sein du *Typed Automata Function Kit*.

Exemplifions ceci sur des automates booléens :

```
$ vcsn-b exp-to-aut --alphabet ab "a*.b" | \
    vcsn-b minimize - | vcsn-b display -
```

donnera l'automate minimal de "a*.b" :



[5] VAUCANSON, `http://vaucanson.lrde.epita.fr/Vaucanson100`.
[6] OLENA, `http://olena.lrde.epita.fr`.

La (2e) minute Lispienne

La légende du fonctionnel (pur)
par *Didier Verna (Enseignant-Chercheur)*

Dans la première minute Lispienne, nous avions fait tomber la légende de la lenteur. Ici, nous nous attaquons à l'idée que LISP est un langage fonctionnel, voire fonctionnel pur. Cette idée n'est pas fausse ; elle est par contre très réductrice. LISP a bien été un langage fonctionnel pur à ses débuts : le tout premier LISP de John Mc Carthy (1958) contenait seulement 7 opérateurs primitifs, tous purs, ainsi que le nécessaire pour construire des fonctions de première classe. Mais le véritable problème est que beaucoup de gens ont tendance à considérer la notion de paradigme de programmation comme quelque chose d'exclusif. Ainsi, dire aujourd'hui que LISP est un langage fonctionnel est extrêmement réducteur, car LISP est en fait un langage multi-paradigme à l'intérieur duquel le fonctionnel peut tout à fait ne jouer aucun rôle.

Dans la suite de cet article, nous mettons le doigt sur trois paradigmes particuliers qui, ajoutés aux aspects fonctionnels, font de LISP un langage souple et expressif.

LISP : un langage impératif LISP contient tout l'attirail impératif classique rencontré dans des langages tels que C ou C++. Pour commencer, le corps d'une fonction LISP peut contenir une séquence d'expressions plutôt qu'une expression unique (il s'agit donc plutôt de « procédures »). Les expressions intermédiaires ne sont alors utiles que pour leurs effets de bord, et l'impureté existe bel et bien en LISP, par exemple grâce aux opérateurs d'affectation, tels le `set!` de Scheme ou le `setf` de COMMON LISP. Notons le souci de propreté de Scheme, chez qui tous les opérateurs impurs sont postfixés par un point d'exclamation. De son côté, COMMON LISP « le pragmatique » nous offre souvent deux versions, pures ou impures, d'un même algorithme (`reverse` / `nreverse`, `remove` / `delete` *etc.*). Le standard COMMON LISP donne également des indications sur ce qu'il est bon ou mauvais de (vouloir) faire. Par exemple, il n'existe pas de version non destructive de la fonction `sort`.

COMMON LISP possède l'« infâme goto ». LISP dispose également de mécanismes de transfert de contrôle non-local au travers des « continuations » de Scheme (plus propres sur le plan théorique) et du `catch` / `throw` de COMMON LISP (plus simple d'implémentation et d'utilisation).

En terme de génie logiciel, retenons qu'un bon programme LISP est souvent impératif (impur) dans les couches basses, là où l'efficacité est de mise et fonctionnel (pur) dans les couches hautes, là où l'abstraction est de rigueur.

LISP : un langage orienté objet COMMON LISP dispose d'une couche objet dont l'expressivité reste à ce jour inégalée : CLOS (COMMON LISP Object System). Plusieurs raisons à cela.

Tout d'abord, la puissance « par défaut » : héritage multiple, multi-méthodes *etc.* sont disponibles immédiatement ; autant de fonctionnalités manquantes dans certains autres langages objets...

Ensuite, la puissance « dynamique » : supposons que vous souhaitiez redéfinir une classe (par exemple, pour en fournir une implémentation plus efficace) alors que l'application concernée est en cours d'exécution. Non seulement CLOS vous permet de le faire, mais cette nouvelle définition est également propagée à toutes les classes dérivées, ainsi qu'à toutes les instances existantes de la classe concernée et de ses sous-classes !

Enfin, la puissance « générique » : prenons C++ pour comparaison. Apprendre l'orienté objet au travers de C++ introduit un biais dans le sens où C++ fournit seulement une *instance possible* du concept objet, lui-même beaucoup plus général. Outre l'absence de multi-méthodes, qui a jamais dit que la notion de classe abstraite était nécessaire ? CLOS ne fournit pas ce concept par défaut, mais vous pouvez le programmer si vous le souhaitez. Qui a jamais dit que le mécanisme de dispatch (sélection de la méthode à appliquer) devait être fixe ? Pourquoi dispatcher sur une seule méthode, pourquoi pas sur une combinaison de toutes celles disponibles ? CLOS vous permet de modifier l'algorithme de dispatch générique, voire même de fournir vos propres mécanismes.

En résumé, CLOS est un système objet *reprogrammable* ; un peu comme si vous pouviez modifier le comportement de C++ (donc le standard...). D'une certaine manière, CLOS est à la fois un système objet (par défaut), et un système de programmation de nouveaux systèmes objets.

LISP : un méta-langage LISP n'est pas seulement un langage de programmation : c'est également un langage de programmation *programmable*. LISP dispose d'un système de macros extrêmement puissant et versatile. Les macros LISP sont en fait des fonctions qui prennent une expression en argument et retournent une nouvelle expression *transformée*. C'est cette dernière expression qui sera finalement utilisée par le compilateur en lieu et place de l'expression d'origine.

Pour comprendre d'où vient la puissance et la spécificité des macros LISP, il faut réaliser que ces macros exécutent véritablement du code LISP pour