La microprogrammation du systeme LELISP: une premiere approche

Matthieu Devin

INRIA
CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tel. (3) 954 90 20

Rapports de Recherche

N° 441

LA MICROPROGRAMMATION DU SYSTÈME LELISP : UNE PREMIÈRE APPROCHE

Matthieu DEVIN

Septembre 1985

LA MICROPROGRAMMATION DU SYSTEME LELISP : UNE PREMIERE APPROCHE

Matthieu Devin

I.N.R.I.A.
Sophia-Antipolis - 06560 Valbonne
Domaine de Voluceau - 78153 Le Chesnay

Résumé :

Nous rapportons ici des travaux effectués dans le cadre du projet VLSI de l'INRIA concernant l'étude de la microprogrammation du langage LLM3, langage de base du système LeLisp* de Jérôme Chailloux. Des gains de vitesse de 15 à 25 % pour l'interprète, et de 15 a 50 % pour les programmes compilés sont obtenus en microprogrammant les parties clefs du système. Ces gains de vitesse sont cependant limités par la lenteur des accès mémoire, générateurs de blocage des microprogrammes, surtout en ce qui concerne les accès à la pile.

Abstract :

We report a first experience of microcoding on the kernel of the Lisp intepreter LeLisp devised by Jérôme Chailloux at INRIA. Microcoding the kernel parts of the system leads to speed improvement from 15 to 25 % for interpreted code, and form 15 to 50 % for compiled code. Those improvements are somewhat limited by processor locks due to memory access slowness. Locks mainly appear when accessing Lisp stack.

Ce travail a pu être mené grâce à la collaboration active de l'équipe PASTIS de Marc Berthod, et surtout celle de son ingénieur système Patrick Cipière.

* LeLisp est une marque déposée de l'INRIA.

1. BUT DE L'ETUDE

Le noyau de l'interprète LeLisp est écrit dans le langage d'une machine virtuelle: la machine LLM3 [Chailloux85]. Le portage de LeLisp sur une machine réelle nécessite l'implantation de cette machine virtuelle [Devin85]. Cette machine est assez proche des machines classiques (VAX 11, MC 68000); l'implantation consiste donc souvent en l'écriture d'un jeu de macros-instructions, traduisant les instructions LLM3 en instructions de la machine cible. Cette manière de porter LeLisp s'avère très performante, tant pour les résultats obtenus (les interprètes portables LeLisp soutiennent la comparaison avec des interprètes dédiés [Chailloux&al.84]), que du temps passé au portage (il a suffit de 3 semaines pour implanter LeLisp sur GOULD/SEL 32).

Cette première technique de portage étant maintenant relativement au point, il est intéressant d'aller plus avant et d'essayer de voir ce que peuvent apporter les machines microprogrammables. La microprogrammation permet en effet de simuler au plus bas niveau la machine LLM3, soit par l'ajout de nouvelles instructions réalisant exactement les instructions LLM3, soit par la microprogrammation des parties clefs de l'interprète (la fonction EVAL notamment). Cette étude est de plus motivée par les recherches actuellement menées dans le projet VLSI pour concevoir une plaque microprogrammée, qui réaliserait la machine LLM3 [Audoire85]. On espère ici mettre en valeur certains problèmes liés à la microprogrammation, notamment les blocages dus aux lectures/écritures dans la mémoire.

2. LES MOYENS

Pour ces premiers essais de microprogrammation nous avons utilisé l'ordinateur Perkin-Elmer 3250, équipé du WCS [PerkinElmer], du projet PASTIS de l'INRIA à Sophia-Antipolis. C'est un ordinateur du même calibre qu'un VAX 11/780, plus rapide pour les opérations flottantes, mais légèrement moins efficace en Lisp.

Le Perkin-Elmer est un ordinateur microprogrammé: l'exécution de chaque instruction provoque un branchement, indexé par le code-op de l'instruction, vers un micro-programme. C'est ce micro-programme qui exécute effectivement l'instruction.

Le système WCS (Writable Control Store) dont est équipé la machine consiste en une mémoire RAM rapide de 2K mots (soit 2K microinstructions), dans laquelle on peut stocker de nouveaux microprogrammes. On lance ces sous programmes grâce à une instruction spécialisée de la machine. Il faut noter que ces programmes ne remplacent pas les micro-programmes qui exécutent les instructions de la machine mais viennent en supplément: on ne rédéfinit pas le jeu d'instructions de la machine. L'instruction permettant de lancer les sous-

programmes n'autorise de plus que 16 points d'entrée différents, le choix des microprogrammes doit donc être judicieux. Il faut aussi tenir compte du fait que cette instruction coûte déjà 3 microinstructions: le coût de lancement d'un micro-programme n'est donc pas négligeable.

2.1. LES MICRO-INSTRUCTIONS

Les microinstructions permettent d'exécuter parallèlement des opérations internes au microcode et des lectures/écritures dans la mémoire.

Les opérations internes au microcode sont soit des opérations de calcul entre les registres (24 registres rapides généraux sont disponibles), soit des lectures/écritures dans la mémoire WCS (le temps de lecture d'un mot est ici 420ns, soit 80 ns de mieux que dans la mémoire centrale). Les opérations de calcul usuelles (transfert, addition, soustraction) ne prennent en général qu'un seul microcycle (soit 260 ns).

Les lectures/écritures dans la mémoire prennent deux microcycles (500 ns pour lire un mot de 32 bits, soit 4 octets), mais sont menées en parallèle avec les calculs. Ceci signifie que l'on doit effectuer une microinstruction complète en attendant le résultat d'une lecture mémoire, ou avant de lancer une nouvelle écriture mémoire. Si l'on a rien à faire en attendant le résultat d'une lecture le processeur est bloqué. Ce blocage du à la mémoire fait perdre un cycle dans l'exécution du microprogramme.

Dans un microprogramme il est donc habile de commencer une lecture mémoire en même temps que l'on réalise un premier calcul, de réaliser un autre calcul pendant le second cycle que prend la lecture, puis d'utiliser le résultat de la lecture dans une troisième microinstruction.

C'est l'exploitation de ces possibilités de parallélisme qui permet de faire des microprogrammes très efficace, où un faible débit mémoire peut être compensé par l'exécution parallèle d'opérations de calcul.

3. REALISATION, ET MESURES

Nous avons réalisé pas à pas la microprogrammation du système. Les progrès on été enregistrés au fur et à mesure pour permettre de mesurer les gains apportés par chaque étape. Nous donnons au fil de ce rapport les temps obtenus à chaque fois sur un bench mark, peu représentatif de vrais programmes LISP il est vrai, mais qui nous a utilement servi de point de référence.

Ce bench mark est constitué du fameux calcul *(FIB 20)* interprété et compilé

par le compilateur LELISP, c'est à dire en liaison dynamique. Nous donnons pour information les temps obtenus par la compilation en liaison lexicale, simulée par l'écriture à la main du code machine correspondant.

La compilation dynamique préserve la liaison dynamique du paramètre formel de la fonction, et implique donc de lourdes manipulation de pile lors de chaque appel ou retour de la fonction. Dans la compilation lexicale on considère que le paramètre de la fonction est en liaison statique ce qui réduit considérablement les manipulation de pile. Nous donnons vers la fin de ce rapport un bench mark plus complet, montrant les gains obtenus sur des programmes LISP plus significatifs.

Les temps (en secondes) obtenus avant toute microprogrammation sont:

Avant Microprogrammation	
interprété	6.30 s
compilé dynamique	0.949 s
compilé lexical	0.22 s

3.1. PUSH ET POP

La première idée de cette étude était d'ajouter les instructions LLM3 les plus couramment utilisées en tant que nouvelles instructions de la machine. Le hardware ne permettant pas le rajout d'instructions, mais uniquement l'écriture de microprogrammes nous avons écrit deux programmes réalisant les instructions d'empilage et de dépilage PUSH et POP.

Dans le microprogramme PUSH la mis à jour du pointeur de pile peut avoir lieu en parallèle à l'écriture dans la pile, il suffit donc au total de 3 microinstructions pour réaliser cette opération.

PUSH et POP		
	temps	gain
interprété	6.48 s	perte 3%

Malgré l'utilisation du parallélisme dans ces deux microprogrammes, le code final est **moins efficace** que le code initial. Ceci est du au coût de lancement des micro-programmes: ici le micro-programme POP coûte aussi cher que l'instruction de lancement des microprogrammes! Nous sommes donc en dessous du seuil où l'écriture de microprogrammes est rentable, il faut s'attaquer à des instructions plus chères.

3.2. CALL ET RETURN

De même que dans PUSH et POP un peut exploiter le parallélisme assez facilement dans l'instruction CALL: Le pointeur de pile est mis à jour parallèlement à l'écriture de l'adresse de retour dans la pile. Résultat: 4 microinstructions suffisent à réaliser l'opération sans blocage du à la mémoire.

Par contre dans le microprogramme RETURN le parallélisme ne peut pas être bien exploité à cause de certaines limitations du hardware*. Résultat: 3 microinstructions suffisent mais cette mauvaise exploitation du parallélisme résulte en un blocage du processeur en attente du résultat de la lecture du sommet de pile; ceci coûte un microcycle supplémentaire.

Le bench mark montre néanmoins un léger gain d'efficacité, on a donc passé le seuil au dela duquel la microprogrammation devient rentable. Il faut aussi noter que l'on gagne environ 10% sur la taille du code assembleur résultat de l'expansion du code LLM3. Ceci est du à la grande place qu'occupait l'instruction CALL avant microprogrammation.

CALL et RETURN

	temps	précédent	gain
interprété	6.17 s	6.30 s	2 %
compilé dynamique	.893 s	.949 s	6 %
compilé lexical	.21 s	.22 s	5 %

3.3. EVAL

La microprogrammation des instructions LLM3 ne donnant pas de résultats enthousiasmants, il nous a paru, à ce stade, plus intéressant de microprogrammer directement des séquences d'instructions. Nous avons donc écrit un microprogramme qui réalise le début de la fonction EVAL, coeur de l'interprète. La fonction LISP EVAL, consiste donc en un appel direct du microcode.

Voici les résultats obtenus, d'abord en microprogrammant uniquement l'évaluation des atomes (nombres, symboles, flottants, vecteurs et chaînes), puis en élargissant le microprogramme à l'évaluation des listes.

L'évaluation des atomes effectue un test de l'état de trace de l'évaluateur, des tests de types, et un test final pour vérifier que la valeur obtenue n'est pas indéfinie.

---

* On ne peut pas changer le compteur programme et initier le décodage d'une instruction dans la même microinstruction.

L'évaluation des listes fait de plus un test de débordement de pile, extrait la fonction et son FTYPE de la liste, et réalise un branchement indexé par le FTYPE vers des programmes spécialisés dans l'évaluation de chaque type de fonction. Pour des raisons d'efficacité la table des branchements indexés est stockée directement dans la mémoire du microcode plutôt que dans la mémoire centrale.

EVAL (uniquement Atomes)			
	temps	précedent	gain
interprété	5.89 s	6.17 s	4.5 %
compilé dynamique	.893 s	.893 s	0 %

EVAL (avec Listes)			
	temps	précédent	gain
interprété	5.43 s	6.17 s	12 %
compilé dynamique	.893 s	.893 s	0 %

On ne gagne évidemment absolument rien pour le code compilé car celui-ci évite tous les appels à l'interprète.

Le pas suivant de la microprogrammation de la fonction EVAL consiste à remplacer tous les appels à cette fonction dans les structures de contrôle LISP par un appel direct au microcode. Ceci revient à remplacer des appels à un sous programme (CALL EVAL) par un appel du microcode; on économise ainsi deux opérations de pile (CALL et RETURN) pour chaque appel à l'évaluateur. Ce gain est particulièrement appréciable dans le cas où la forme à évaluer est un atome, et mieux, une constante. Du point de vue conceptuel ceci revient à implanter une nouvelle instruction machine, réalisant l'évaluation des formes LISP. On s'approche de ce que l'on peut appeler une machine LISP.

Les gains d'efficacité deviennent appréciables, ce qui confirme le fait que la microprogrammation doit s'attacher à des séquences d'instructions LLM3 plutôt qu'aux instructions LLM3 elles-mêmes.

instruction EVAL			
	temps	précédent	gain
interprété	4.89 s	5.43 s	10 %
compilé dynamique	.893 s	.893 s	0 %

3.4. LA LIAISON DYNAMIQUE

Pour l'étape suivante nous avons donc choisit de microprogrammer les routines LLM3 qui construisent et détruisent les blocs de pile nécessaires à la liaison dynamique LISP. Nous avons donc écrit des microprogrammes correspondant aux points d'entrée UNBIND, CBINDi et CBINDE. Nous avons de plus réalisé, comme pour EVAL, la transformation des appels à ces points d'entrée (par CALL et par JUMP) un appel direct aux microprogrammes. Tout se passe donc comme si l'on avait rajouté les instructions machine correspondantes.

instructions CBIND1/2/3/E			
	temps	précédent	gain
interprété	4.89 s	4.89 s	0 %
compilé dynamique	.693 s	.893 s	22 %

instruction UNBIND			
	temps	précédent	gain
interprété	4.72 s	4.89 s	3 %
compilé dynamique	.484 s	.693 s	30 %

Les points d'entrée CBINDi sont utilisés uniquement par le compilateur, c'est pourquoi on ne gagne rien dans le code interprété. Les gains pour le code compilé sont ici très intéressant; ceci est du au fait que la fonction FIBONNACCI compilée ne fait pas grand chose hormis des appels fonctionnels, et donc passe le plus clair de son temps dans la construction et la destruction des blocs de pile.

3.5. REVUE DES BLOCAGES MEMOIRES

Nous avons finalement opéré une réécriture globale de tous ces microprogrammes, en essayant d'éliminer les blocages dus à la lenteur des accès à la mémoire centrale. Nous avons donc essayé d'utiliser les possibilités de parallélisme au maximum.

Plusieurs points de blocage ne peuvent cependant être éliminés, notamment dans les programmes de construction des blocs de pile, qui ne font pas grand chose hormis des accès mémoire dans la pile. On ne peut donc pas exploiter à fond les possibilités de parallélisme des microinstructions car il n'y a assez de calculs à faire en attendant les résultats des lectures/écritures mémoire. Pour ce qui est de la liaison dynamique on est donc limité par la lenteur des accès à la pile. L'utilisation d'un cache de pile (les N premiers sommets de pile deviennent des registres, N > 100), comme prévu dans la plaque LLM3, semble donc

gagnante.

Elimination des Bloquages		
	temps	gain
interprété	4.72 s	25 %
compilé dynamique	.401 s	58 %
compilé lexical	.21 s	5 %

4. BENCH-MARK COMPLET

Voici les temps obtenus après revue des microprogrammes, sur plusieurs programmes LISP réels.

Résultats Globaux		
programme	*gain interprété*	*gain compilé*
(FIB 20)	25 %	58 %
(ACK 3 4)	22 %	62 %
(TAK 18 12 6)	21 %	54 %
Compilation LISP	28 %	
Esterel (CEYX)	--	20 %
Système Expert (CEYX)	18 %	20 %
Système Expert (LISP)	14 %	38 %
Matching Symbolique	14 %	18 %

La compilation LISP correspond à la compilation de l'environnement LISP standard (éditeur, trace, compilateur..). Celle-ci est réalisée en grande partie par du code compilé.

Esterel [Berry&al.84] est un langage de programmation parallèle du au projet Parallélisme et Synchronisation de l'Ecole de Mines de Paris, dirigé par Gérard Berry. Les mesures correspondent à une moyenne effectuée sur les temps de compilaton de deux versions de la montre CASIO, et du jeu de reflexes.

Le Système Expert CEYX est un système expert de classification de Galaxies du à Catherine Granger du projet PASTIS [Granger&al.85]. Le Système Expert LISP est ce même système expert, après suppresion de toute la mécanique CEYX, et réorganisation complète (la différence de temps avec le précédent n'est pas uniquement due à la suppression de CEYX). Les mesures correspondent à la classification d'une cinquantaine de galaxies.

Le Matching Symbolique est aussi du à Catherine Granger [Granger84]. Les mesures correspondent à la mise en correspondance d'un modèle d'empreintes digitales avec une empreinte réélle.

Il ressort de ce bench-mark que l'on peut compter sur un gain de vitesse de 15 à 20 % pour les programmes CEYX, et de 15 à 40 % pour les programmes LISP.

5. CONCLUSION

Ces premiers essais de microprogrammation de la machine LLM3 montrent que l'on peut facilement atteindre des gains de 15 à 40 % pour des programmes LISP interprétés ou compilés.

Pour les programmes interprétés le gain des performances est pour moitié du à la microprogrammation de la fonction EVAL, que l'on pourrait pousser plus loin (notamment pour les fonctions prédéfinies à nombre fixe d'arguments). La microprogrammation de la routine de destruction des blocs de pile n'apporte qu'un faible gain (3%) : cette partie n'est donc pas critique pour l'interprétation.

Pour les programmes compilés nous nous sommes surtout attachés à réduire le coût de la liaison dynamique. Pour des petites fonctions ce coup est très important (nous gagnons jusqu'à 60% pour la fonction TAK) et la microprogrammation des routines de constructions des blocs de pile était donc nécessaire. Ce travail devient par contre inutile dans le cadre d'un LISP lexical.

Dans un LISP lexical (et LELISP en sera bientôt un), il faudrait plutôt s'attacher à microprogrammer des séquences de l'interprète appelées à *runtime* par le code compilé. Par exemple le coeur des fonctions de parcours de liste (MAPxxx) ou la recherche des symboles dans l'OBLIST (READ, SEND, SYMBOL). On a ainsi le double avantage de gagner du temps pour l'interprétation et pour la compilation des fonctions LISP.

PROBLEMES LIES A LA MICROPROGRAMMATION

La difficulté majeure rencontrée dans la microprogrammation de la machine LLM3 est la lenteur des accès à la mémoire centrale. Dans le microcode on en vient à concevoir les accès en mémoire centrale comme des entrées-sorties, c'est à dire prenant un temps non négligeable et devant être bien synchronisées. La réécriture d'une première implantation de la routine CBIND1, pour supprimer les blocages mémoire a permis de gagner 17 % sur notre fonction d'essai !

Une autre difficulté est l'impossibilité de tracer l'exécution des microprogrammes, qui sont ininterruptibles. Deux fois nous sommes restés coincés dans

le microprogramme (dans une boucle infinie), et il a été nécessaire de couper l'alimentation du CPU pour relancer la machine. La mise au point était cependant facilitée par la possibilité de modifier le microcode de la machine interactivement, grâce à un driver UNIX spécialisé, et à un microassembleur interactif (qui a été construit en LISP au début de l'étude).

6. ANNEXES: LES MICROPROGRAMMES

Nous donnons ci-dessous un listing commenté de quelques microprogrammes utilisés pour implanter la machine LLM3 sur Perkin-Elmer.

6.1. LES INSTRUCTIONS DE PILE

6.1.1. RETURN

Le point d'entrée RETURN

```
 Lecture de l'adresse de retour au sommet de pile
      l       mar,sp,dr4          ; 0(sp) --> rmdr
  Mise à jour du compteur programme
      lx      cloc,rmdr,#$12      ; cloc <- adresse rmdr, suite en 12 :
  Mise a jour du pointeur de pile et décodage de l'instruction suivante
      a       sp,sp,ysi,ird       ; sp a jour, decode l'instruction suivante
```

6.1.2. CALL

Le point d'entrée CALL

```
Préparation de l'écriture de l'adresse de retour
 et lecture de l'adresse du sous programme appelé
      s       mar,sp,ysi,dr4ib    ; mar <- sp-4, rmdr <- adresse
      lx      wmdr,cloc,#$10      ; adresse retour -> wmdr, suite en 10
mise a jour du compteur programme
 et écriture de l'adresse de retour dans la pile
      l       cloc,rmdr,dw4       ; retour -> 0(sp), cloc <- adresse
Mise a jour du pointeur de pile
 et décodage de l'instruction suivante
      l       sp,mar,ird          ; sp mis a jour, decode la suite
```

6.2. EVALA1

Le microprogramme suivant évalue l'objet LISP contenu dans le registre A1, et rend sa valeur dans ce même registre. Si l'objet est un atome (symbole, nombre, vecteur ou chaîne) on n'effectue aucune manipulation de pile.

Test de la trace :
comparaison des registres evalst et rnil

```
        s       (),evalst,rnil,()
        baleq   (),#$945,()          ;  c'est nil --> EVALAN
```

Appel de l'évaluateur non microcodé en cas de trace
#$942 (02) c'est pas tracé

```
        bal     (),#$972             ;  CALL EVALT
        l       (),(),()
        l       (),(),()
```

Tests de types sur l'objet a évaluer :
1- Est-ce un flottant ?
oui: EVAL termine immédiatement, sans avoir utilisé la pile.
#$945 (05)

```
        nil     (),a1,#$852          ;  un flottant 31 bits ?
        baleq   (),#$947,ird         ;  oui : retour immédiat
```

2- Est-ce une cellule de liste ?
oui: il faut évaluer une fonction
#$947 (07)

```
        s       (),rbcons,a1,()      ;  Etes-vous CONS ?
        balng   (),#$957,()          ;  oui --> EVALIS
```

3- Est-ce un symbole ?
oui: on va prendre sa valeur
non: c'est une constante, EVAL termine immédiatement
(toujours sans manipulation de pile)

```
        s       (),rbsymb,a1,()      ;  est-ce un SYMBOLE ?
        balng   (),#$94b,ird         ;  oui : on l'évalue, non : la suite
```

Evaluation des symboles :
Lecture de la CVAL en parallèle chargement du symbole UNDEF dans un registre
#$94B

```
        l       mar,a1,dr4           ;  Lit la CVAL de a1
        ai      mr0,rnil,32          ;  met UNDEF dans mr0
        l       a1,mdr,()            ;  a1 <- cval(a1)
```

La CVAL est-elle égale au symbole UNDEF ?
oui: appel la routine d'erreur
non: EVAL termine (la pile n'a pas bougé)

```
        s       (),a1,mr0,()         ;  a1 =? UNDEF
        baleq   (),#$950,ird         ;  oui : erreur, non : décode la suite
```

Appel de la routine d'erreur si CVAL=UNDEF

#$950

```
        l       a2,mar,()           ; a2 <- adresse symbole
        s       mar,sp,ysi,()       ; il faut simuler le CALL
        l       wmdr,cloc()
        l       sp,mar,dw4
        l       a1,a2,()            ; ?? pas undef dans a1 ??
        lii     cloc#$860           ; BRA EVALERA8
        l       (),(),ird
```

Evaluation des listes

Il faut ici simuler un CALL EVAL en empilant une adresse de retour.

#$957 (17) CALL EVALIS

```
        s       mar,sp,ysi,()       ; pour simuler le CALL on fait PUSH CLOC
        l       wmdr,cloc()
        l       sp,mar,dw4
```

Test de débordement de la pile (en cas de récursion infinie par ex.)

```
        lii     0,#$851             ; Test de débordement de pile
        sdec    (),sp,0,#$960       ; pas de debordement
```

La pile déborde, appel de la routine d'erreur

#$95C (1C)

```
        lii     cloc#$861           ; BRA ERRFS
        l       (),(),ird
```

Point d'entree pour évaluer les fonctions qui ne sont pas des symboles

#$95E

```
        lii     cloc#$862           ; BRA EVALFV
        l       (),(),ird
```

Extraction de la fonction: le CAR de la liste à évaluer

#$960

```
        l       mar,a1,dr4          ; lit le car de la forme
        l       forme,a1,()         ; sauve la forme dans 'formé
        l       a2,rmdr,i4dr4       ; lit le CDR, A2 <- (CAR A1)
        l       a1,rmdr,()          ; A1 <- (CDR A1)
```

Test du type de cette fonction

on appelle EVALFV si la fonction n'est pas un symbole

```
        s       (),rbcons,a2,()     ; bcons-a2
        balng   (),#$95E,()         ; si bcons-a2 <= 0 -> a2 est un cons
        s       (),rbsymb,a2,()     ; bsymb-a2
        balg    (),#$95E,()         ; si bsymb-a2 > 0  -> a2 pas un symbole
```

Extraction des champs FVAL et FTYPE

```
        ai      mar,a2,#$8          ; prepare lecture FVAL(A2)
        s       sp,sp,ysi,dr4       ; prépare le PUSH et lit FVAL(A2)
        l       mar,sp,()           ; pour le push
        l       wmdr,rmdr,dw4       ; PUSH FVAL(A2)
        ai      mar,a2,25           ; lire le ftype
        l       (),(),dr1           ; lire l'octet
```

Branchement indirect indexé sur le FTYPE

```
        l       a3,rmdr,()
```

```
        ai      mr0,a3,#$870        ;  la table d'indirection
        il      clocmr0,()          ;  dans le pc
        l       (),(),ird
```

Point d'appel de l'évaluateur avec trace

```
#$972 (32)              CALL EVALT
        s       mar,sp,ysi,()
        l       wmdr,cloc()
        l       sp,mar,dw4
        lii     cloc#$863           ;  CALL EVALT
        l       (),(),ird
```

7. REFERENCES

References

[Audoire85] Louis Audoire, "La Plaque Microprogrammée LLM3," Rapport de recherche en préparation, INRIA, Rocquencourt (D).

[Berry&al.84] Gérard Berry and Laurent Cosserat, "The Esterel Synchronous Programming Language and its mathematical semantics," Rapport de recherche No 327, INRIA, Rocquencourt (D).

[Chailloux85] Jérôme Chailloux, "La Machine LLM3," Rapport de recherche a paraître, INRIA, Rocquencourt (D).

[Chailloux&al.84] Jérôme Chailloux, Matthieu Devin, and Jean-Marie Hullot, "LELISP: A Portable and Efficient LISP System," in *4th ACM conf. on LISP and Functional Programming*, Austin, Texas (Août 1984).

[Devin85] Matthieu Devin, "Le Portage du Système LELISP: Mode d'Emploi," Rapport Technique No 50, INRIA, Rocquencourt (D).

[PerkinElmer] Perkin Elmer, *Perkin Elmer 3240 Programming Manual*, 1970.

[Granger84] Catherine Granger, "Matching a Scene to a Model using Symbolic and Geometric reasoning," in *ECAI 1984: Advances in Artificial Intelligence*, ed. T. O'Shea (1984).

[Granger&al.85] Catherine Granger and Monique Thonnat, "Un système de vision fondé sur une méthode structurelle de classification," Cognitiva 1985, Paris (D).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique