Applications Development Environment for the CAD Information Systems Design

Alexandre Gavrilov
Cybernetics Department,
MEPhI
Moscow, Russia
bs-avg@east.ru

Alexandre Saevsky

Moscow, Russia

Abstract

The paper describes a developing environment for AutoLisp. The Developers Environment goals, structure and theoretical base are discussed. The presented Project Subsystem organisation provides the powerful optimisation ability. The foundations and results of this optimisation are present. The subjects of this paper are the results of the evaluation model constructing, the architecture development and the toolkit system design and implementation. One of the main problems of the application systems design is the choice of the reliable toolkit. AutoCAD is a widely spread CAD environment. Now AutoCAD includes the powerful graphic editor, SQL-level data bases toolkit, GUI features of DCL and so on. From version Release-12, one of the supporting platforms is 'Microsoft Windows'.

1. Introduction

The subjects of this paper are the results of the evaluation model constructing, the architecture development and the toolkit system design and implementation.

One of the main problems of the application systems design is the choice of the reliable toolkit. AutoCAD is a widely spread CAD environment. Now AutoCAD includes the powerful graphic editor, SQL-level data bases toolkit, GUI features of DCL and so on. From version Release-12, one of the supporting platforms is 'Microsoft Windows'.

Proceedings of the Workshop on Computer Science and Information Technologies CSIT'99
Moscow, Russia, 1999

But the AutoCAD customisation language i.e. AutoLISP has no GUI interface, development support for programmers and applications packager tools. It's a reason for the AutoLISP Developers Environment (DevEn) creation.

2. *AutoLISP* Developers Environment

2.1 Developers Environment goals.

This environment is intend to AutoLISP developers, which elaboration result is delivery packages. The main part of their work is the LISP-programs written and debugging, the packages constitution. It's a reason, that the DevEn consists of the text editor, debugging and errors diagnostic tools (static and dynamic) and delivery packages creation means, which includes AutoLISP compiler and run time support system. Now we present general statements of the Developers Environment organisation and more detail consider the application packages creation.

The Developers Environment is implemented as a powerful integrated functional systems, which contains two dialects of LISP. Subset Common LISP named LEX with the object oriented subsystem was applied as the internal implementation language. AutoLISP evaluation system was supplied as a parallel to the Standard AutoCAD's interpreter.

2.2 Theoretical foundations

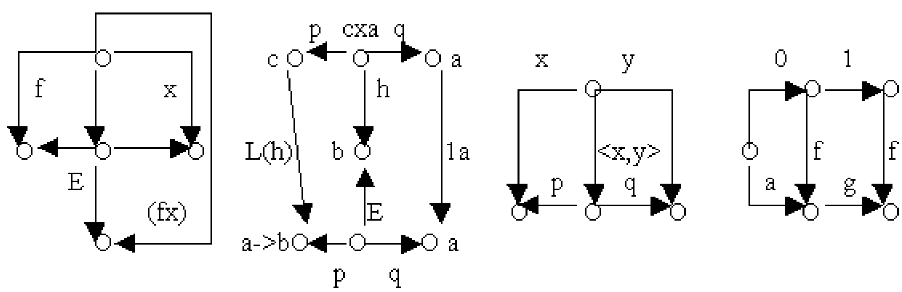
The categorical combinatory logic is chosen for the evaluation model. Its embodiment is a categorical abstract machine (CAM). In our case the traditional variant of CAM is extended. The call-by-name and freeze-evaluate means are added to Curien's model. LISP dialects compilation is based on the mathematically strong translation scheme (see Figure 1). This property provides the semantic correctness of the executing code and ensure the safe optimisation facility for source programs packaging.

```
-Operation------------------------------------------------------------------------------
Application      Abstraction      Dotted pair        Recursion

-Functional Language----------------------------------------------------------------------
(E f x)            (lambda x h(x))     (cons x y)            f = if (x=0)
(car '(x . y))          then a
(cdr '(x . y))          else g(f(x-1))

-Lambda-calculus-------------------------------------------------------------------------
(f x)                  Lx.h(x)        [x,y] = Lxyr.rxy       f = Y(Lx.(= x 0) a
p[x,y] = x, p = CIK       (g (f (- x 1)))),
q[x,y] = y, q = CI(KI)  Y = Lf.((Lx.f(x x))

-Catgorical theory-----------------------------------------------------------------------
```




```
-CAM-codes ---------------------------------------------------------------------------------
(push) | [f]p | (swap)      (cur [h(x)](p.x))     (push)|[x]p|(swap)  (push dum cons push if
| [x]p | (cons app)                                      |[y]p|(cons)     (push)|[x](p.f)|(swap
                                                                           quote 0 cons eq)[a](p.f)
                                                                           (push)| [g](p.f) |(swap
                                                                           push cdr swap push) |
                                                                           [x](p.f)|(swap quote 1
                                                                           cons sub cons app cons
                                                                                   app) wind cdr)
```

Figure 1. The categorical support of the functional languages

2.3 DevEn structure

Any means which supports the development of AutoCAD applications are: Complete LISP dialects emulation, Source Debugger, Syntax Checker, Integral Text Editor, Compiler and runtime support kernel, Inspector of functional objects and AutoCAD entities and a rich set of debugging features, like expression's value watch, Backtrace dialog, easy source code access (see Figure 2). The integration of our system with the 'Microsoft Windows' environment gives the ability of the traditional object and interface means of this environment using. So, we implement the powerful toolkit for the AutoLISP application systems constructing.

3. Project subsystem organisation

At this chapter we detail consider the application packages organisation. The idea of 'project' is chosen for the large source code programs organisation. The project description is placed in one text file and contains main information for the package creation and usage. The native AutoLISP semantic modifications are localised in this place. There are three considerable parts of the project subsystem usage:

1) the construct stage, which includes the program codes typing and static checking of its correctness by interactive DevEn;

2) the build stage, which includes the delivery packages building with compiling, merging and optimisation features by AutoLISP Compiler;

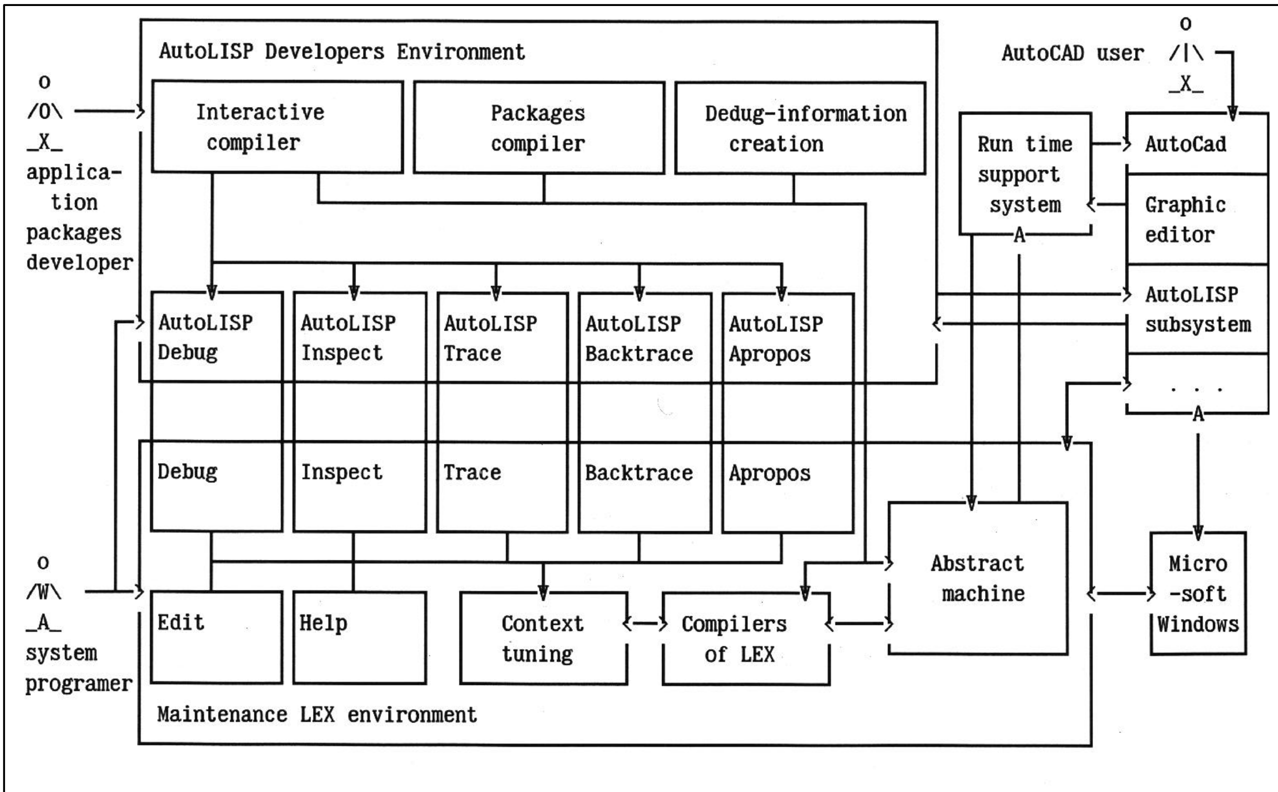


Figure 2. AutoLISP Developers Environment Structure

3) the run time - the execution of the source and compiled programs by Run-Time Support system.

3.1 Construct stage

Main directions of construct stage are:

1. Edit project files with browsing
   - easy access to file
   - easy access to symbol definition
   - easy access to symbol references
   - find string
   - replace sting
   - find S-expression with pattern matching
   - replace S-expression
   - analyse based lexical colors
2. Check and Analyze features
   - syntax (system and user declared)
   - special data base creation
   - static data base analysis
   - function calls correctness
   - external (unresolved) references and variables
   - unreferenced definitions and variables
   - main statistic
   - alphabetic ordered list of local variables
   - safe optimisation recommendations
   - compile specific
3. Debug
   - file consistence
   - debugger
   - data bases toolkit
4. User Interfaces
5. Documentation support features.

3.2 Build stage

More important features, that are provided by LISP compiler, are:

- speedup of loading and running
- function calls linkage
- package building
- fast loading files concatenation
- variables names collision extraction
- names security
- consult, provide, require features

- function calls substitution

- strip of surplus information

3.3 Run time stage

The run time stage must be provide the AutoLISP package execution, where the package may take one of the following forms:

- only lisp files,

- only fast load format (fas) compiled files, where each fas-file relate to one source lisp file,

- single fas-file, which includes the whole package.

4. Code Optimization

4.1 Reasons

One of the main lacks of the natural AutoLISP is absence of the tools for powerful packages creation. DevEn provides the ability of the standalone powerful applications creation by the optimising compiler The main optimisation features are:

- the functions calls linkage;
- the variables localisation;
- the names dropping;
- the fast-load code generation;
- the inline code expansion;
- the continuations-depended compilation;
- the peep-hole optimisation;
- the source codes transformation;
- the multiple literals creation.

The correctness of the optimising codes execution is based on the strong mathematical foundation. The optimisation increases the application speed 4 - 10 times as much, and modern packaging forces the user and commercial value of the elaboration.

4.2 The optimisation correctness analysis

We should note, that as for other compilers, turning on program optimisation may bring bugs to the valid code. The builder performs partial analysis of correctness of program optimisation. Thus it extracts the information about:

- explicit function calls
- explicit SETQ and DEFUN variable assignments
- explicit value references
- localisation of variable assignments and value references
- QUOTED argument of EVAL form
- QUOTED function argument for APPLY and MAPCAR
- static (explicitly typed) action-strings in ACTION_TILE function calls
- exporting to ACAD information (described using special pragmas and declarations) to avoid drop names

It does not account:

- the dynamically build code that can be called than using EVAL, APPLY, MAPCAR and LOAD
- setting dynamically supplied variables by SET
- dynamic (program evaluated) action-strings in ACTION_TILE function calls

All the optimisations leads in general to changing program semantics. It may lead to losing access to program functions and symbols from the outer application and from the AutoLISP console. Thus some functions available from console in interpreter mode turns unknown in compiled mode. Also some functions may stay available from outer user, but redefining the functions will not lead to losing all references to old function definition.

However, the compiler intend to preserve the behaviour of the project components *relatively* to the other project parts. Note, that if your code do not use the AutoLISP expressions listed before, the ANALYSER will supply the full information for program optimisation correctness.

4.3 Checking optimising consistency

The compiler always check optimising consistency. The consistency conditions deals either with project or with current top-level expression. The optimising revoke conditions are divided to hard-revoke conditions and safe-revoke conditions. The hard optimising revoke conditions deals with the compiler abilities and rude semantic changes.

The compiler always checks this conditions and issue the warnings when the user recommended an optimisation but the compiler cannot apply it because of a revoke condition. This conditions are briefly listed below.

LINK conditions

The compiler links system (build-in) lisp function when they were not redefined and not bound and not assigned anywhere in the project.

The compiler links user function calls if they were defined using **`DEFUN`** once (and only once) in the entire project and all the function calls argument number fits to the found definition.

DROP conditions

The compiler tries to drop function name only if all the function calls should be linked to the function definition.

The compiler does not drop function name for a function definition if

- The symbol called by name
  - exported to ACAD by **`export-to-acad`** - pragma slot.
  - referred in **`ACTION_TILE`** action string
  - referred as quoted function argument for **APPLY** or **`MAPCAR`** somewhere in the project.

LOCALISE conditions

The compiler does not localise a variable in bound lists in **`DEFUN LAMBDA`** and **`FOREACH`** expressions if :

- The variable has a non-local reference (or assignment) to the variable within the outer top-level expression.
- The variable is called by name (as it was defined in previous subsection)
- The variable symbol appears in the function name position somewhere in the outer top-level expression.

Safe-revoke conditions and SAFE optimising mode

Besides the hard optimising revoke conditions, the current compiler allows to treat the stronger revoke conditions that we mentioned as safe-conditions. That can decrease the amount of actually applied optimisations but provide a better proof of the code correctness. The safe-conditions deals with uncertain effects that can take place while running a program and lead to failure of optimised program while the source code were just valid.

An example

One can use a function symbol FOO and define it by DEFUN and then link it. Let the FOO being also assigned somewhere in the code using (setq FOO <expr>). That may change the code semantic and may not change it. Thus, if the assigned value <expr> is intended to be used as function body, the code semantic will deviate when compiling without safe-mode. The safe mode will revoke the linking and the initial semantic will be preserved. On the other hand, if the identical names are used only independently, the safe-mode applies the overmuch care of program semantic and possibly lacks the code efficiency. However, the safe-mode is on by default and that is recommended mode when you face the compiler first time.

Link safe-revoke conditions:

- The symbol is bound as parameter anywhere in the project
- The symbol is bound as auxiliary variable and referenced as value anywhere in the project
- The symbol is explicitly assigned somewhere (by SETQ statement).

Drop safe-revoke conditions:

The symbol is referenced as value.

Link safe-revoke conditions:

- The variable has a non-local reference or assignment to the variable within the project
- The variable is called by name (as it was defined in previous subsection).

5. Optimisation Foundations

5.1 General Notion

prj $\in$ **PRJ** That is a DevEn project considered now as a sequence of source files and (optionally) required entities

sfile $\in$ **SRCFILES** (Lisp source files)

prjconsult $\in$ **PROJECTCONSULT**

expr $\in$ **EXPR** (e_1, e_2 ...)

s $\in$ **SYM** (f, x, y, …)

n $\in$ **N** (i, j, k, l...)

Fin(***Set***) - set of finite subsets of ***Set.***

Bind-expr **`:= (DEFUN …) | (LAMBDA …) [FOREACH]`**

(Meta)variable f is proposed for function symbols, and x,y.. are used for bound symbols.

We say that e_1 is an valuable **true** sub-expression of e_2 and denoted it $e_1 \subset^* e_2$ iff

e_2= **`(DEFUN`** ***`name`*** **`(…) …`** e_1**`…) or`**

e_2= **`(LAMBDA (…) …`** e_1**`…) or`**

e_2= **`(SETQ`** … v_1 e_1**`…) or`**

*e_2= **`(EVAL ’`**e_1**`) or`**

*e_2= **`(MAPCAR ’(LAMBDA (…) …`** e_1 **`…) …) or`**

*e_2= **`(APPLY ’(LAMBDA (…) …`** e_1 **`…) …) or`**

e_2= **`(COND … (`**e_1 **`…) …) or`**

e_2= **`(COND … (…`** e_1**`) …) or`**

e_2= **`(FOREACH`** ***`var`*** e_1 **`…) or`**

e_2= **`(FOREACH`** ***`var …`*** e_1 **`…) or`**

*e_2= **`(’(LAMBDA (…) …`** e_1 **`…) …) or`**

e_2= **`(`*****`f`*** **`…`** e_1 **`…)`** `where` ***`f`*** Ï ***`special-form-name`***

`special-form-name` **`:= QUOTE | FUNCTION | PRAGMA | TRACE | UNTRACE | …`**

`and, or, if` are <u>not</u> considered here as special forms

We say that e_1 is an valuable sub-expression of e_2 and denoted it $e_1 \subset e_2$ iff e_1 , e_2 belongs to the transitive closure of $\subset^*$ - relation.

LISP source files are considered as sequences of tl-expressions, that is top-level expressions. Thus we drop comments & illegal data. If the parser meets reader error we treat this file as having the last top level equal to **error-value**.

Free (expr) :=

{expr} if expr $\in$ **SYM**

{} if expr is atom and not symbol

Free(e) if expr = **(EVAL (QUOTE** e**))** or expr = **(FQUOTE** e**)**

Free(e) U Free(e_1) if expr = **(APPLY (QUOTE** e**)** e_1**)** and e is **(LAMBDA …)**

$\cup$,(e_1, $e_{2\ldots}$ e_n) Free(e_i) U Free(e) if expr = **(MAPCAR (QUOTE** e**)** $e_1 \ldots e_n$**)** and e is **(LAMBDA …)**

{} if expr = (QUOTE e_1)

$\cup$,(e_1, $e_{2\ldots}$ e_n) Free(e_i) \ {$x_1, \ldots x_k$} if expr = **(DEFUN *name* (**$x_1, \ldots x_k$**)** $e_1 \ldots e_n$**)**

$\cup$,(e_1, $e_{2\ldots}$ e_n) Free(e_i) \ {$x_1, \ldots x_k$} if expr = **(LAMBDA (**$x_1, \ldots x_k$**)** $e_1 \ldots e_n$**)**

($\cup$,(e_1, $e_{2\ldots}$ e_n) Free(e_i) \ {x})U (Free (expr)) if expr = **(FOREACH** x expr $e_1 \ldots e_n$**)**

$\cup$,(e_1, $e_{2\ldots}$ e_n) Free(e_i) U Free(fun) if expr = **(**fun $e_1 \ldots e_n$**)**

5.2 Ground functions

Bound function gets the set of symbols bound in the expression

IBound-as-parameter (expr) =

{$x_1 \ldots x_k$} iff expr = **(DEFUN *name* (**$x_1 \ldots x_k$ […]**) …)**

{$x_1 \ldots x_k$} iff expr = **(LAMBDA (**$x_1 \ldots x_k$ [/ …]**) …)**

{} otherwise

Bound-as-parameter (expr) = $\cup$,e $\subset$expr **IBound-as-parameter**(e)

IBound-as-aux(expr) =

{$x_1 \ldots x_k$} iff expr = **(DEFUN *name* (… /** $x_1 \ldots x_k$**) …)**

{$x_1 \ldots x_k$} iff expr = **(LAMBDA (…/** $x_1 \ldots x_k$**) …)**

{x} iff expr = **(FOREACH x …)**

{} otherwise

Bound-as-aux(expr) = $\cup$,e $\subset$expr **IBound-as-aux**(e)

Bound (expr) = **Bound-as-aux**(expr) U **Bound-as-parameter**(expr)

Funcall-argnum function gets the number of arguments in function calls in *expr* for a given function *f*

I-Funcall-argnum (expr, f) =

$\perp$ if expr is not **(**f **…)**

$\perp^*$ (reference without argument number)

if expr = **(**{**APPLY | EVAL |MAPCAR**} **'**f **…)**

n if expr = **(**f $e_1 \ldots e_n$**)**

$\perp < \perp^* < n < T$

sup (n,m) = T iff $n \neq m$

Funcall-argnum (expr,f) =Sup,e $\subset$expr (**I-Funcall-argnum** (e, f))

Defun-argnum function gets the number of arguments in function definitions in *expr* for a given function *f.*

I-Defun-argnum (expr, f) =

$\perp$ if expr is not **(DEFUN *f* (…) …)**

n if expr = **(DEFUN *f* (**$e_1 \ldots e_n$ [/ …]**) …)**

Defun-argnum(expr,f) =Sup,e $\subset$expr (**I-Defun-argnum** (e, f))

Defun-number function gets the number of **DEFUN**-statements in *expr* for a given function *f.*

I-Defun-number(expr,f) =

1 if expr = **(DEFUN *f* (…) …)**

0 otherwise

Defun-number(expr,f) = Σ,e $\subset$expr (**I-Defun-number** (e, f))

Value-reference function gets the set of symbols referenced as value in *expr*.

I-Value-references (expr) =

{s} if s=expr

{f} if expr = **(EVAL '**f**)**

{} otherwise

Value-references (expr) = $\cup$,e $\subset$expr**I-Value-references** (e)

Assigned function gets the set of symbols assigned in *expr*.

I-Assigned(expr) =

{$s_1 \ldots s_k$} if expr = **(SETQ** s_1 e_1 … s_k e_k**)**

{s} if expr = **(SET** 's e**)**

{s} if expr = **(FOREACH** s e**)**

{} otherwise

Assigned(expr) = ∪,e ⊂expr**I-Assigned**(e)

Non-local-Assigned function gets the set of symbols assigned in *expr*, not bound in the assigning scope

Non-local-Assigned(expr) = ∪,bind-expr ⊂expr(**Assigned**(bind-expr) ∩ **Free** (bind-expr)}

Non-local-value-referenced function gets the set of symbols referenced in *expr* not bound in the reference scope.

Non-local-value-referenced(expr) = ∪,bind-expr ⊂expr (**Value-referenced**(bind-expr) ∩ **Free** (bind-expr)}

5.3 Certain equations

Now we extend the predicate definitions to the projects

Bound-as-aux $_{prj}$ = ∪,tl ⊂ PRJ**Bound-as-aux**(tl)

Bound-as-parameters $_{prj}$ = ∪,tl ⊂ PRJ**Bound-as-parameter**(tl)

Funcall-argnum $_{prj}$(f) =Sup,tl ⊂ prj (**Funcall-argnum** (tl, f))

Defun-argnum $_{prj}$(f) =Sup, tl ⊂ prj (**Defun-argnum** (tl, f))

Defun-number $_{prj}$ (f)= ∑,tl ⊂ prj **Defun-number**(tl,f)

Value-references $_{prj}$ = ∪,tl ⊂ prj**Value-references** (tl)

Assigned $_{prj}$ = ∪,tl ⊂ prj**Assigned**(tl)

Non-local-Assigned$_{prj}$ = ∪,tl ⊂ prj**Non-local-Assigned** (tl)

Non-local-value-referenced$_{prj}$ = ∪,tl ⊂ prj **Non-local-value-referenced** (tl)

The following predicates will make the following clauses less verbose and more apparent.

DEFINED $_{prj}$ = {x | **Defun-number** $_{prj}$(x) > 0}

ONCE-DEFINED $_{prj}$ = {x | **Defun-number** $_{prj}$ (x) = 1}

CALLED $_{prj}$ = {x | **Funcall-argnum** $_{prj}$ (x) ≠ ⊥}

CALL-BY-NAME $_{prj}$ =

- {call to function appears in not-link pragma context}
- **EXPORT-TO-ACAD** pragma exists
- function name appears in DEFUN and fits to: **AUTO-EXPORT-TO-ACAD-PREFIX**
- {expr = **(**{**APPLY** | **MAPCAR**} **'**f **…)** found as a valuable true sub-expression of any top-level expression in the project.

EXTDEFP $_{prjclosure}$ = **DEFINED** $_{prjclosure}$

INIT-SYS-FUN - the set of initially defined functions familiar to AUTOLISP compiler

FUNCTIONS = **DEFINED** U **INIT-SYS-FUN**

5.4 Forbidding Optimize oracles

Not-sys-fun =

{f∉ INIT-SYS-FUN}
U DEFINED
U Assigned
U Bound
U EXTDEFP
U {f∈INIT-SYS-FUN | Funcall-argnum(f) ⊄ Init-argnum(f)}

Cannot-link-p $_{prj}$ (f) =

if f∈ INIT-SYS-FUN
then f∉ Not-sys-fun
else
{f∉ONCE-DEFINED
or {**DEFUN** and current tlf are in different modules}
or Funcall-argnum(f) = T
or Funcall-argnum(f) ≠ Defun-argnum(f)
;; The safe condition follows
or f∈Bound-as-parameter
or (f∈Bound-as-aux and f∈Value-referenced)
or f∈Assigned }

Cannot-drop $_{tlf}$ (f)=
{exists function call and **DEFUN** in different modules}
or Cannot-link-p $_{prj}$ (f)
or {f ∈ CALL-BY-NAME}
;; Safe condition follows
or (if {**DEFUN** f == current tlf} Non-local-value-referenced (f) else Value-referenced (f))

Table 1. The Compiler testing results

System	Test	LOAD (s)	DERIV (s)	TAK (s)	TAKL (s)	TIM (s)	FIB (s)
ACAD 12R	lsp	11	182	93	141	99	171
DevEn Optimise Compiler	lsp	14	89	46	74	44	41
	fas	2	36	11	16	9	17

The not-drop-from-aux-p predicate is always "safe"! This predicate appears …

Not-Drop-from-aux-p $_{tl}$ (f) =

Value-referenced $_{prj}$
U Assigned $_{prj}$
U CALL-BY-NAME $_{prj}$
U EXTDEFP $_{prjclosure}$
U EXTREFP $_{prjclosure}$

Cannot-localise-p $_{tl}$ (f) =

Non-local-assigned $_{tl}$
Non-local-value-referenced $_{tl}$
DEFINED
;; Safe conditions follows
U CALLED
U EXTDEFP
U EXTREFP
U Non-local-assigned
U Non-local-value-referenced
U CALL-BY-NAME

6. The optimisation results

The analyse of the compiler efficiency was based on the Gabriel's tests set. This set includes the large structures handling tests, mathematical tests, recursive calls etc. The results of testing are represented in Table 1. The analyse shows, that the DevEn packages aren't yield to analogic systems by main testimonials.

Conclusion

Thus, the usage of the project notion in AutoLISP Developers Environment provides the increase of the comfort and efficientence of the large application packages design in AutoCAD, the speed up of delivery products and code security for technology know-how safety.

Acknowledgement

I wish to thank Prof. V. Wolfengagen for his assistance in preparing the final version of this paper.

References

1. *AUTOCAD Release 13 Customization Guide*, Autodesk, 1994.
2. Banaitre J., Jones S., LeMetayer D. "Prospect for functional programming in software engineering", *ESPRIT-302*, 1991.
3. Cousineau G., Curien P.-L., Mauny M., Suarez A. "Combinateurs Categorique et Implementation des Langages Fonctionelles". - *LNCS N242*, 1986, p. 85 - 103.
4. *Create and Debug LISP Code with ADE*. Cadence. November, 1993.
5. Eriksen L. *AutoLISP Tools. A look at Four Debugging Products*. Cadence. March, 1994.
6. Field A.J., Harrison P.G. *Functional Programming*.- Addison - Wesley publishing Company, 1988.
7. Gabriel R.P. *Performance and Evaluation of Lisp System*, Mit- Press. 1985.
8. Gavrilov A.V. "New Developers Tools for the CAD Information Systems Design". In *Third International Workshop on Advances in Databases and Information Systems (ADBIS'96)*, MEPhI Publishing, Moscow, 1996.
9. VitaL Lisp. "Development, debugging, delivery tools for *Basis Software*, Exton, PA 19341, 1995.