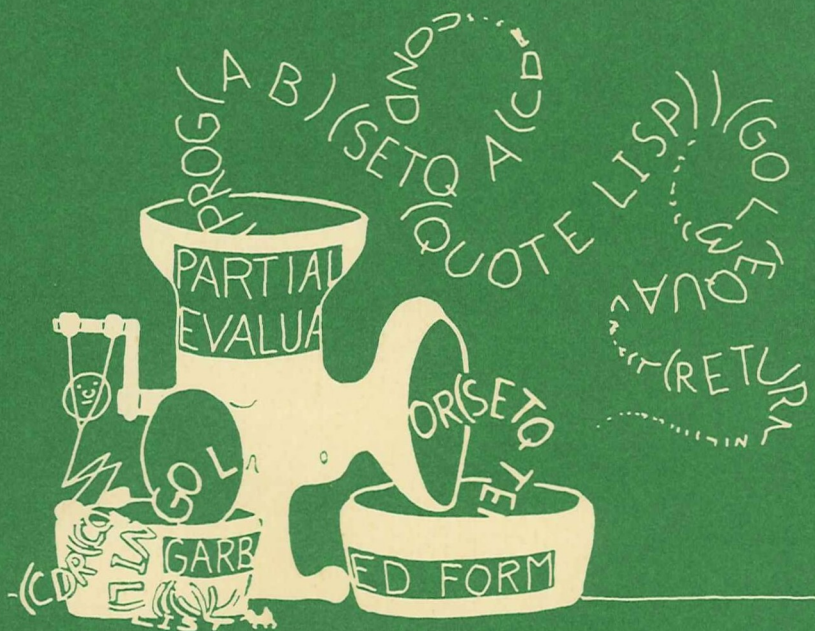


**Linköping Studies in Science and Technology Dissertations  
No 14**

# A PROGRAM MANIPULATION SYSTEM BASED ON PARTIAL EVALUATION



by Anders Haraldsson

**Department of Mathematics**  
**Linköping University, S-581 83 Linköping, Sweden**  
**Linköping 1977**

**A PROGRAM  
MANIPULATION SYSTEM  
BASED ON  
PARTIAL EVALUATION**

**by Anders Haraldsson**

**Akademisk avhandling  
som för avläggande av filosofisk doktorsexamen  
kommer att offentligt försvaras  
i lärosal BDL 6, hus B, Valla  
fredagen den 27 maj 1977 kl 10.15.**

**A PROGRAM  
MANIPULATION SYSTEM  
BASED ON  
PARTIAL EVALUATION**

**by Anders Haraldsson**



**Department of Mathematics  
Linköping University, S-581 83 Linköping, Sweden  
Linköping 1977**

ISBN 91-7372-144-1

Vimmerby Tidnings Tryckeri 1977

## ABSTRACT

Program manipulation is the task to perform transformations on program code, and is normally done in order to optimize the code with respect of the utilization of some computer resource. Partial evaluation is the task when partial computations can be performed in a program before it is actually executed. If a parameter to a procedure is constant a specialized version of that procedure can be generated if the constant is inserted instead of the parameter in the procedure body and as much computations in the code as possible are performed.

A system is described which works on programs written in INTERLISP, and which performs partial evaluation together with other transformations such as beta-expansion and certain other optimization operations. The system works on full LISP and not only for a "pure" LISP dialect, and deals with problems occurring there involving side-effects, variable assignments etc. An analysis of a previous system, REDFUN, results in a list of problems, desired extensions and new features. This is used as a basis for a new design, resulting in a new implementation, REDFUN-2. This implementation, design considerations, constraints in the system, remaining problems, and other experience from the development and experiments with the system are reported in this paper.

Key-words: program manipulation, partial evaluation, program optimization, LISP, beta-expansion, macro-expansion

This research was sponsored in part by the Swedish Board for Technical Development (STU) under contract Dnr 4732 and by the Swedish Natural Science Research Council (NFR) under contract Dnr 2654.

## ACKNOWLEDGEMENTS

This thesis has been carried out at Datalogilaboratoriet, Computer Science Department at Uppsala University and at the Informatics Laboratory at Linköpings University.

Thanks go to my thesis supervisor Erik Sandewall for starting up this project in program manipulation, for discussions about this topic, for his advice in preparing this thesis and for reading the manuscripts. Thanks also to Jim Goodwin, Jerker Wilander and Östen Oskarsson for reading parts of my manuscript and giving valuable comments and criticisms, and to Rene Reboh and Jaak Urmi for discussions and all other colleagues for various help during this work.

I also want to thank Lennart Beckman, Erik Sandewall and Östen Oskarsson for their participation in the early REDFUN work. Lennart Beckman was responsible for the REDCOMPILE program, Östen Oskarsson for the the experiments with the GIP/GUP program and Erik Sandewall for the initial code for REDFUN. Thanks to Tore Risch for his PMG-program used in this work and Jim Goodwin for writing the iterative statement which was an excellent program to test my system.

Thanks also to Nan Strömberg for typing this thesis, Olle Willén for illustrations and to Anthony Skeat for correcting my English.

This thesis is dedicated to my family, Eva and Karin, who have served and supported me during this work.

## CONTENTS

1, INTRODUCTION.....	13
2, PROGRAM MANIPULATION AND PARTIAL EVALUATION	
2.1 Program manipulation.....	17
2.2 Manipulation of source code.....	18
2.3 Program analysis.....	19
2.4 Program manipulation system.....	20
2.5 Partial evaluation.....	21
2.6 Related work in program manipulation and partial evaluation.....	24
3, AN OVERVIEW OF THE REDFUN PROJECT	
3.1 A design iteration model for the REDFUN project.....	27
3.2 First version.....	29
3.3 Second version.....	30
4, THE REDFUN SYSTEM	
4.1 Summary of the REDFUN report.....	33
4.1.1 The REDFUN program.....	33
4.1.2 REDFUN in the PCDB application.....	40
4.1.2.1 Generation of storedef-procedures.....	40
4.1.2.2 Example from the axiom compilation step..	47
4.1.3 REDFUN in the GIP/GUP experiment.....	51
4.1.4 Conclusions from the usage of the REDFUN program.	54

4.1.4.1 Experiences from the PCDB application....	54
4.1.4.2 Experiences from the GIP/GUP experiment..	57
4.1.5 The REDCOMPILE program.....	59
4.2 A study of manipulation in the WTFIX file.....	61
4.2.1 Conclusions from the study.....	64

## 5. DESIGN ALTERATIONS FOR REDFUN-2

5.1 New features in REDFUN-2.....	65
5.1.1 Extended range of information about variable values.....	65
5.1.2 Handling of values from expressions.....	66
5.1.3 Systematic handling of side-effects, especially assignments.....	67
5.1.4 Extraction of variable properties from the code..	69
5.1.5 Reduction of <u>prog</u> -expression .....	70
5.1.6 Other features.....	72
5.2 Principal changes in the design of the new system re- lative to REDFUN.....	73
5.2.1 The design of the program structure.....	73
5.2.2 Value-descriptors.....	74
5.2.3 The q-tuple.....	75
5.2.4 Semantic procedure.....	77
5.2.5 Redesign of the <u>prog</u> -reducer.....	77
5.2.6 Redesign of the substitution package .....	77
5.2.7 Reduction in contexts.....	78

## 6. THE REDFUN-2 SYSTEM

6.1 Overview.....	81
6.2 Extended range of information about variables.....	83



6.2.1 A-list.....	83
6.2.2 Datatypes.....	84
6.2.3 Reduction of value-descriptors.....	86
6.3 Quoted-expressions.....	90
6.4 Computing values of forms.....	93
6.4.1 Application of function.....	93
6.4.2 Computing values from conditionals and logical functions.....	95
6.4.3 Pseudo-computations.....	96
6.4.4 The <u>tryapply</u> -procedure.....	101
6.4.5 Summary of the semantic procedure.....	105
6.5 Extraction of variable properties.....	106
6.5.1 True- and false-branches.....	106
6.5.2 The <u>truectxt</u> - and <u>falsectxt</u> -elements.....	107
6.5.3 Example.....	108
6.5.4 Pure functions and <u>ctxtfn</u> -procedures.....	110
6.5.4.1 Example.....	110
6.5.5 Logical functions and conditionals.....	112
6.5.5.1 <u>And</u> - and <u>or</u> -expressions.....	112
6.5.5.2 <u>Cond</u> -expression.....	113
6.5.5.3 Algorithms.....	114
6.5.6 Own "datatypes".....	117
6.6 Handling of side-effects.....	119
6.7 Variable assignments.....	120
6.7.1 Assignments in arguments to a function of pure, expr or sideexpr-type.....	120
6.7.2 Assignments inside conditionals and logical func- tions.....	122

6.7.2.1	Algorithm.....	122
6.7.2.2	<u>Setq</u> 's reducer.....	124
6.7.2.3	The :ADDVALUE-descriptor.....	124
6.7.2.4	The :SETQVALUE-descriptor.....	125
6.7.2.5	Specialization, replacement and extension of a value-descriptor.....	126
6.7.3	Transferring of assignment information.....	128
6.7.4	Assignments by the function <u>set</u> .....	128
6.7.5	Global variables.....	129
6.8	Extended function class authority.....	131
6.8.1	Function classes.....	131
6.8.2	Functions belonging to several classes.....	133
6.8.3	The function class of <u>cons</u> .....	133
6.9	Reduction of <u>prog</u> -expression.....	136
6.9.1	Assignments in a <u>prog</u> -expression.....	136
6.9.1.1	<u>Prog</u> -expressions without loops.....	137
6.9.1.2	<u>Prog</u> -expressions with loops.....	140
6.9.1.3	Some problems occurring in this method...	142
6.9.2	Postprog-transformations.....	143
6.10	Opening of functions.....	146
6.10.1	Open classes.....	146
6.10.2	Examples of open classes.....	146
6.10.2.1	Beta-expansion.....	147
6.10.2.2	Open specialization.....	148
6.10.2.3	Closed specialization.....	149
6.10.3	Automatic procedure to decide open class.....	149
6.10.4	Substitution package.....	152
6.11	Collapsers.....	155
6.12	Reduction in contexts.....	156

6.12.1 Reduction in contexts.....	156
6.12.2 Contexts.....	157
6.12.3 Context table.....	158
6.12.4 Example of a form in the different contexts.....	159
6.12.5 Changes of contexts.....	161

## 7. A NEW APPLICATION AND NEW EXPERIMENTS WITH THE REDFUN-2 SYSTEM

7.1 A partial evaluator as a macro expander.....	165
7.2 Macro-expansion in the INTERLISP-system.....	166
7.3 Discussion of <u>map</u> -functions.....	167
7.4 Experiments with the iterative statement.....	169
7.4.1 Description of the application.....	169
7.4.2 Performance.....	171
7.4.3 Detected problems.....	173
7.4.3.1 Unrolling of <u>prog</u> -expressions.....	173
7.4.3.2 Beta-expansion.....	175
7.4.3.3 Interrelation between reduction and analysis of code.....	176
7.4.3.4 <u>Set</u> -expressions.....	177
7.4.4 Other amendments.....	178

## 8. CONCLUSIONS

8.1 Summary of this thesis.....	181
8.1.1 The REDFUN-2 program.....	181
8.1.2 The feasibility of the proposed design.....	182
8.1.3 Some weaknesses in the proposed design.....	182
8.1.3.1 Handling of <u>go</u> -statements.....	182
8.1.3.2 Handling of side-effects.....	183

8.1.3.3 Problems in collapsers and postprog- transformations.....	184
8.1.4 Complexity.....	185
8.1.5 Generality.....	185
8.1.6 Reliability.....	186
8.1.7 Efficiency.....	187
8.2 Directions for futher work.....	188
8.2.1 The REDFUN-2 program.....	188
8.2.2 The REDCOMPILE program.....	189
8.2.3 Theoretical work.....	190
APPENDIX I	
Generating <u>storedef</u> -procedures in the PCDB application.....	193
APPENDIX II	
Changes of contexts - an example.....	201
APPENDIX III	
<u>Map</u> -functions - examples.....	205
APPENDIX IV	
The experiment with the iterative statement.....	217
APPENDIX V	
Program code for some central functions in REDFUN-2.....	235
APPENDIX VI	
Examples from chapter 6 run through the REDFUN-2 program.....	247
REFERENCES.....	261

## 1. INTRODUCTION

This thesis describes a system, called REDFUN-2, which performs partial evaluation, beta-expansion (opening of functions) and other operations on program code written in the LISP language. The system is based on an older version, the REDFUN program, and the experience from a number of experiments with that system. This has been reported in "A Partial Evaluator and its Use as a Programming Tool" (BEC76). For those who want to penetrate this thesis in depth it is recommended first to read that report, which in more detail gives the background and the underlying ideas for this project, although some of it is also found in chapter 4 in this report.

Chapter 2 gives an overview of program manipulation and related tasks which we think can be done by such systems. It includes a survey of related work. It also introduces the concept partial evaluation.

Chapter 3 gives a model which describes the development of the REDFUN-project as a design iteration process and gives some background of the old work.

Chapter 4 is a recapitulation of what was said about the REDFUN system in the report mentioned above. For the reader who is acquainted with the report the first section can be skipped.

Chapter 5 gives a list of the new proposed features and extensions to be included in the new version called REDFUN-2, and the design considerations taken to implement them.

Chapter 6 is a rather detailed description of the actual implementation of the system. This part can be skipped by those who are not really interested in the details of the new implementation.

Chapter 7 describes the use of partial evaluation as a tool to perform macro expansion. The new system is tested on a real program, an implementation of the iterative statement found in CLISP. From a very general program which executes all variants of statements, a specialized version, directly corresponding to a given iterative statement, can be generated.

Chapter 8 gives an evaluation of the REDFUN-2 system and a summary of the experience gained from this work and gives some directions on further work with this system.

Appendix I gives an example from the PCDB-application which illustrates the partial evaluation technique when functions are opened on several levels. This example is recommended for those who want to get an idea of how partial evaluation works applied to a real example.

Appendix II gives an example of how contexts (see 6.12) are changed during the reduction of an expression.

Appendix III shows some examples of how map-functions can be treated in order to automatically generate different versions of them.

Appendix IV gives the LISP code for the executor in the iterative statement and shows a number of examples run through the REDFUN-2 system.

Appendix V gives the LISP code for some of the central functions in the REDFUN-2 system and give some examples of simplifications rules used by the system.

Appendix VI gives the output from some examples from chapter 6 run through the system.

To read this report one needs to have a rather good knowledge about LISP. If you have not, we think only sections 2 and 3 and some of the examples in the appendices are readable and understandable.

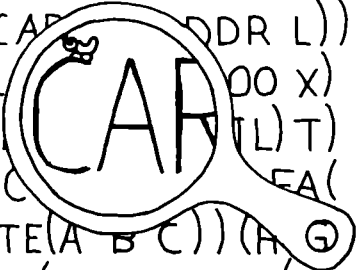
Some of the examples in the text are such that no programmer will ever write such code. The reader may wonder why bother about them. However, the main purpose of this system is not to reduce manually written code, but to reduce code which appears when code is either automatically generated or has previously been manipulated, for example when functions have been opened. In such situations it is necessary also to consider such odd cases.

The implementation of REDFUN-2 was first done on INTERLISP/360-370 (INT75) at Uppsala Datacentral, but the program has recently been moved to INTERLISP/20 on the DEC-20 at Informatics laboratory at Linköpings University. The system works on programs written in the INTERLISP/360-370 dialect, although only minor changes would be necessary to take care of code written in the INTERLISP/20 dialect.

```

ND((ATOM(CAR L)))(
ONS(CAR CDDR L))
(RETU DO X)
ETQ (L) T)
))(T(C FA(
(QUOTE(A B C))(A G)
(MAPC(FUNCTION(LA

```



"PROGRAM ANALYSIS"



## 2. PROGRAM MANIPULATION AND PARTIAL EVALUATION

### 2.1 PROGRAM MANIPULATION

By program manipulation we mean the task of performing various operations on program code in order to modify it in some way. This includes

- compilation of program code to machine code.
- translating by a preprocessor of, for example, a more problem-oriented notation to program code.
- translating a program from one dialect to another.
- macro expansion, normally done on assembly code level, but which can also be done on high level code.
- beta-expansion or opening of procedures, replacing a procedure call by the procedure definition, in which formal arguments are substituted for actual ones.
- optimization of a program to make it more efficient. Examples of such optimization transformations are recursion removal, removal of invariant expressions from a loop, combining loops etc.
- propagation of variables to replace a variable by a constant.
- partial evaluation where as much calculation and simplification in the code as is possible is done before running it. This can be done if for example we know the value of the predicate in an if-statement, in which case the if-statement can be replaced by either the true or the false branch of the statement. This is particularly useful together with opening of procedures and propagation of variables.

Most work has been concentrated on optimizing program code in compilers and in Allen and Cocke (ALL72) a catalog of such optimization transformations can be found. During recent years, however, a growing interest has started, especially with regard to research, in performing other kinds of program manipulation.

## 2.2 MANIPULATION OF SOURCE CODE.

Our primary interest is manipulation on the source code level, so techniques for compiling and optimization by compilers will not be discussed in this text. We consider that it is much more difficult to make reliable compilers if too many transformations and optimizations are done in the compilation step. It is normally impossible for the typical user to know when and where optimizations have been performed in the code. Some serious bugs have been introduced by erroneous optimizations, e.g. when subexpressions in loops, containing side-effects, have been moved outside the loop. If these transformations were done on the source code level it could give the user a better chance to see and understand what transformations had been done.

A common feature of all program manipulation programs is that they process other programs. This means that programs must be represented in the computer in a form which can be manipulated. For this reason it is an advantage to work in a language where we have program and data equivalence such as LISP, so from this point we are mostly concerned with program manipulation in a LISP environment, especially the INTERLISP (TEI74) dialect. Some of the results here are also applicable to conventional languages, but some of the more interesting methods are however more difficult to realize in such languages. Implementing a partial evaluator, where we need an interpreter for the language, is not as easy there.

Some interesting areas where we can find the need for program manipulation on the source code level are listed below.

- To be used during program development. During the last years new methods of developing programs have been intensively discussed. We have structured programming (DIJ70) and step-wise program refinement (WIR71) among similar ideas. The programs must be wellstructured and developed in a more systematic way, reflecting the problem and the algorithms better than before. Smart coding and other tricks are prohibited. A program developed in this way will however not always meet its operational requirements, it must be more efficient. After the program has been finally tested it must go through an optimizing phase. A way of performing this could be to let the programmer in dialog with an optimizer supply information about the program and the optimizer which transformations may suitably be performed. The user could state what special cases can occur in the program, the properties of the data and other information from the application and computer environment.
- To be used in a step before compilation. Instead of complicating the compiler by making it smarter much can be done on the source code level before the compilation. Macro expansion and recursion removal are such transformations which today are carried out by the INTERLISP compiler.
- To be used when translating programs from one dialect to another. It ought to be possible to describe the differences between the dialects in some way and let the translation be made more or less automatically.

- To be used when programs are generated automatically. It seems easier first to generate a more stereotyped version of a program, often not efficient, and in a second step optimize it. The generator will then often be more clearly written and easier to maintain.
- To be used in a "smart" editor, where the user has at his disposal more advanced commands like "substitute the variable x by the constant 5 and carry out all the simplifications which are now possible".
- To be used when specializing a general program. Suppose that in an application we want to run a highly parametrized program several times with the same parameters set to the same constant values each time. We would then like to extract from the general program a specialized version which could be run more efficiently in this application. This can in some cases be nicely done by partial evaluation.

This list can be extended with more areas, but it reflects some of the ideas we have had in mind throughout our work in the area of program manipulation.

### 2.3 PROGRAM ANALYSIS

Closely related to program manipulation is program analysis, where we are interested in extracting information from the program. A performance analysis is desirable and tells where in the program optimization is necessary. Analysis of program flow, variable scope, side-effects etc. must often be performed before a transformation is allowed to take place. An invariant subexpression with side-effects is not normally allowed to be removed from a loop. It is of course also useful to perform program analysis when programs have to be documented.

## 2.4 PROGRAM MANIPULATION SYSTEM

One way to go ahead is to write one program manipulation program (PMP) or program analysis program (PAP) for every application. The disadvantage of this is that we shall duplicate a lot of work. All PMP's and PAP's must know how to scan the program code, in this case LISP code and must know the semantic properties of special functions, such as cond and selectq and other special functions defined by the user. Different manipulation tasks need the same basic transformations and analysis to be performed. Two differently written PMP's or PAP's will probably have different conventions, so it may be impossible to run them together.

This leads us to the conclusion that it would be desirable to have a more integrated program manipulation system (PMS), which contains the basic tools to perform analysis and manipulation of programs. This is an analogy to all the formula manipulation systems which have been developed during the last ten years.

Knuth (KNU74) describes in an article on structured programming the need for such a system. In his enthusiasm he writes:

"The programmer using such a system will write his beautifully structured, but possibly inefficient, program; then he will interactively specify transformations that make it efficient. Such a system will be much more powerful and reliable than a completely automatic one. We can also imagine the system manipulation measurement statistics concerning how much of the total running time is spent in each statement, since the programmer wants to know which parts of his program deserve to be optimized, and how much effect an optimization will really have.

The original program P should be retained along with the transformation specifications, so that it can be properly understood and maintained as time passes. As I say, this idea certainly isn't my own; it is so exciting I hope that everyone soon becomes aware of its possibilities."

So it is clear that it is an important task to try to construct program manipulation systems and to learn more about program transformations, both in theory and practice.

## 2.5 PARTIAL EVALUATION

Partial evaluation is a technique, which in its simplest case can be described as follows:

Suppose P is a procedure of n arguments  $(x_1, x_2, \dots, x_n)$  and that the values for the first m arguments are  $(c_1, c_2, \dots, c_m)$ . A new procedure P' can now by partial evaluation be generated such that

$$P'(x_{m+1}, \dots, x_n) = P(c_1, \dots, c_m, x_{m+1}, \dots, x_n)$$

for all  $x_i, i = m+1, n$

The procedure P' is generated in such a way that in the procedure body of P the variables  $x_i$  are replaced by the constants  $c_i$  and as many calculations as can possibly be performed in the procedure body are done. Functions, which do not have or depend on side-effects and which will now get constant arguments, will be applied to those arguments and the function call can be replaced in the body by the value of that application. If a predicate in a conditional has a known value, either its true or false branch can be discarded.

In a more general definition we can also allow other knowledge about the program to be used in order to perform the partial evaluation.

A short example may be appropriate here although the rest of this report contains numerous examples of partial evaluation.

Suppose foo is defined in an Algol-like language as

```
procedure foo(x,y,z);
integer x,y,z;
begin
integer a;
a := sin(x);
if y < 3 then fie(a+z) else fie(a*2+z/2);
print(a)
end;
```

and if  $x = 1$  and  $y = 5$  a procedure foo' can be generated such that

```
foo'(z)=foo(1,5,z)
```

The procedure foo' will be

```
procedure foo'(z);
integer z;
begin
fie(1.682942+z/2);
print(0.841471)
end
```

We assume that the procedure fie does not use the variables x, y and a freely. Following operations have to be performed:

- to calculate  $\sin(1)$
- to reduce the if-statement to its false branch
- to replace all occurrences of a to its value and to calculate  $a*2$
- to remove x and y from the argument list and a as a local variable

## 2.6 RELATED WORK IN PROGRAM MANIPULATION AND PARTIAL EVALUATION

---

In the domain of conventional programming languages not many projects have been reported. A FORTRAN-to-FORTRAN optimizing compiler is described in (SCH72), in which FORTRAN code is improved through flow analysis, dead variable elimination, code rearrangement, common subexpression elimination, and constant propagation. Loveman (LOV76) describes a large number of source-to-source transformations. He has developed his own language, Penultima 75, which contains most of the facilities found in other "well-structured" languages, in which the transformations can be tested. He also reports on a Language Laboratory, which is a tool designed to assist in the development of optimization techniques for high level programming languages.

In the LISP oriented domain a large project is in progress by Wegbreit and Cheatham (CHE72, WEG75a, WEG76) in the field of automatic programming. They have built a system ECL, in which a LISP-Algol like language EL/1 is defined. Program manipulation is an important part of that system. They have a program analysis system (WEG75b) from which they can derive closed-form algebraic expressions from simple LISP-program execution behavior. The analysis establishes performance goals and these goals will direct the processing of transformations, which are carried out by local simplifications, partial evaluation of recursive functions, abstraction of new recursive function definitions from recurring subgoals and generalization of expressions required to obtain compatible subgoals. Darlington and Burstall (DAR72, DAR73) describe a system which automatically improves programs. The main transformations involved are recursion removal, elimination common subexpressions and combining loops, replacing procedure calls by their bodies and reusing discarded list cells. Later work (BUR75) describes transformations performed on expressions in a form of first order recursion equations.



The projects mentioned here, except the FORTRAN-to-FORTRAN compiler, concern more or less idealized programs. These work either in a very pure LISP or in individually developed languages, such as EL/1 and Penultima 75. On the full INTER-LISP level we can mention work by Teitelman (TEI73, TEI74), such as CLISP, where a translation is made from an Algol based notation to LISP expressions. Our work at Datalogilaboratoriet in Uppsala and Linköping has been of use in the entire language, even if we can only handle a subset of it. We can mention some of our programs, REDFUN (BEC76) for partial evaluation, the basis for this report, REDCOMPILE (BEC76) a kind of compiler for REDFUN, REMREC (RIS73) for recursion removal, PMG (RIS74) a generator for program manipulation programs and FUNSTRUC (NOR72) for analysis of call-structure and variable usage in LISP programs.

Partial evaluation has been used by several researchers for a variety of purposes and in Beckman et al (BEC76) some of them are listed with references. Among new usages of this technique is a project lately reported by Wegbreit (WEG76). His partial evaluator takes expressions such as

$$P(Q_1(y_1), \dots, Q_k(y_k), y_{k+1}, \dots, y_n)$$

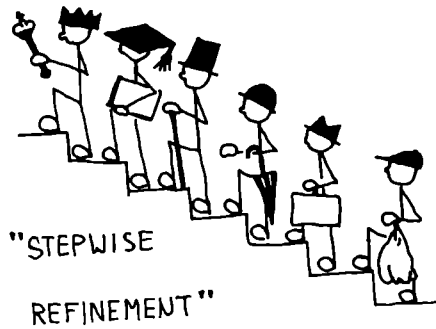
where  $Q_i$  is a defined function (with one argument for simplicity) and constructs a new function  $P'$  such that

$$P'(b_1, \dots, b_k, b_{k+1}, \dots, b_n) = P(Q_1(b_1), \dots, Q_k(b_k), b_{k+1}, \dots, b_n)$$

That is,  $P'$  is a variant of  $P$ , specialized to the case where the first  $k$  parameters are known to be computed by  $Q_1, \dots, Q_k$ .

Other related research areas are program verification automatic programming and a lot of work in programming languages and in compiler construction. From program verification we can find methods which for instance prove that a set of transformations applied to a program will result in an equivalent program.

Systems for automatic programming will certainly need program manipulators when the program code is to be generated. Of interest is of course the development of new programming languages which are more suitably constructed in order to facilitate verification and manipulation of programs and methods for describing syntax and semantics for programming languages.



### 3. AN OVERVIEW OF THE REDFUN PROJECT

#### 3.1 A DESIGN ITERATION MODEL FOR THE REDFUN PROJECT

The REDFUN programs have been developed through a design iteration process. This means that a first version is implemented and then tested, changed and extended successively while new ideas, experienced and methods are developed. After a while the program tends to be rather unstructured. A more systematic design is then performed and a new version of the program is implemented. This program is now in its turn tested etc and the process goes on. This model of program development is found when difficult tasks are tackled, and when no complete theory or methods for solving them exists in advance. This is the typical case for programs developed in artificial intelligence and related research areas. This reasearch methodology in computer science is described by Sandewall (SAN77).

Figure 1 is a schematic presentation of the iteration process for REDFUN. A row in the figure represents one iteration cycle and a box a step inside an iteration. The capital letters will be used throughout the present report to indicate where in the iteration process we are at that moment.

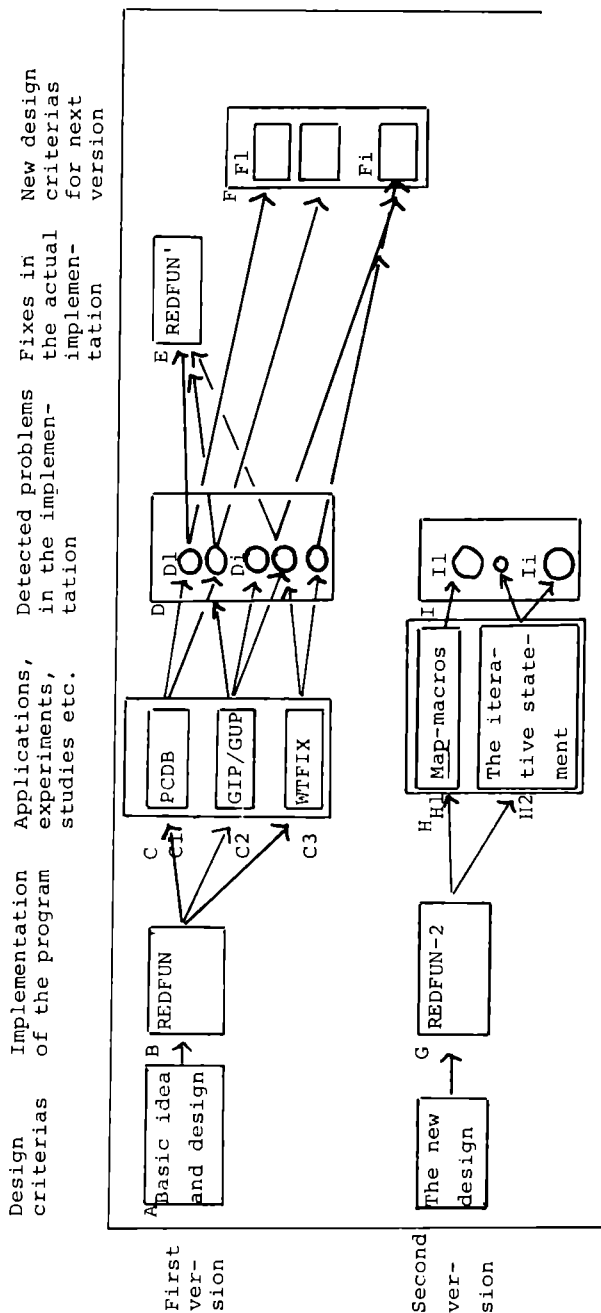


Figure 1 Development of the REDFUN program

### 3.2 FIRST VERSION

The first REDFUN program was developed in connection with the work on the PCDB system (SAN71 HAR73, HAR74). It is a system for maintaining a data base of formulas in predicate calculus and can be seen as a program generator. From a specification of a problem stated in predicate calculus PCDB generates LISP procedures for storing and accessing assertions in a data base. From axioms it generates procedures performing the corresponding deduction. The program generator used a partial evaluation technique together with beta-expansion (replacing procedure calls by procedure bodies after necessary substitutions) and some other operations on programs, and this was carried out by REDFUN. This first version was designed for needs in PCDB and worked quite well, although some minor problems occurred, which resulted either in updates in the system or rewriting the code in PCDB so REDFUN could process it. REDFUN could operate on code which was written in a clean way, with no disturbing side-effects and assignments and prog-expressions restricted to its format etc. This early work corresponds to the steps A, B and C1 in fig 1, and also to some of the contributions in D.

The next step was to try to apply REDFUN to a program, which was not especially designed or influenced by REDFUN. A program GIP/GUP, performing general input and output of information on property lists, was chosen as candidate. The program was very parametrized and the task for REDFUN was to generate a specialized version of the general program GUP in an application, where a number of these parameters had known values. This new experiment pointed out a number of weaknesses of the system. Many problems occurred because of arbitrary use of assignments and prog-expressions. A lot of changes and extensions were made to the system, so it worked sufficiently well in this application, although some of the fixes were performed more or less temporarily. In fig. 1 these steps are represented by boxes C2, contributions to D and box E.

In connection with the PCDB work to speed up the generation of procedures a "compiled" version of REDFUN was built, called REDCOMPILE. If REDFUN operates on a program P and generates a specialized version SP we can use REDCOMPILE to translate the program P to a generator version GP, which now in its turn can generate the specialized version SP. The generation in the later case is faster then in the first one, but the translation of P to GP naturally takes time. If several SP's are to be generated, total time will however be saved, which was the case in the PCDB application. If REDCOMPILE is put in the schemata of fig 1 instead of REDFUN, that program has passed through the steps A, B and C1.

The use of the partial evaluation technique and this early part of the REDFUN project have been reported in "A Partial Evaluator, and its Use as a Programming Tool" published in the Artificial Intelligence Journal (BEC76). In sections 4.1-4.4 of this report some parts of that article will be recapitulated. In the schemata in fig 1 this corresponds in principle to the first iteration cycle (steps A to E). The study in step C3 was not included in the paper.

### 3.3 SECOND VERSION

The main work described in this thesis is the next iteration step of the REDFUN program. It started by going back to step C, where a study (C3) of the appropriateness of REDFUN and the partial evaluation technique was performed to see if it could act as a more intelligent editor. The problem was to extract from a large file WTFIX, only those parts of functions there, which served a special purpose, ie the functions had to be specialized for a special application. The task was partly one of partial evaluation, e.g. when we knew that a variable in this application had a special value or values and that we could go through the code and make reductions. Some results from this study are reported in section 4.2.

Problems detected so far (D) are found in sections 4.1.4 and 4.2.1. Among the most important problems to solve were the handling of arbitrary assignments and side-effects, and to maintain additional information about the variables used during the reduction.

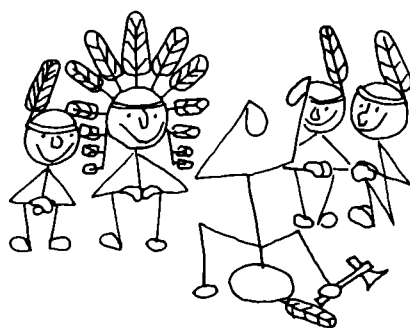
A new design was performed of the system (F) to solve some but not necessarily all, of the detected problems. This was carried out among others by introduction of the q-tuple, whereby more than one value can be returned from a function, and semantic procedures, by which semantic properties about functions to operate on can be given to the system.

The new implementation (G), called REDFUN-2, is described in section 6. A new application to let the partial evaluator expand macros before compilation of LISP programs, and some new experiments of this (H) are described in section 7. New detected problems (I) after these experiments are also reported there.

In section 8 we discuss some directions in which the work on this system, the next iteration cycle, can proceed.

The following table shows where in this report the respective steps in fig 1 are described

step	sections
A }	4.1.1
B }	
C1	4.1.2
C2	4.1.3
C3	4.2
D	4.1.4, 4.2.1
E	-
F	5.1-5.2
G	6.1-6.12
H1	7.3
H2	7.4
I	7.4.3



"RED FUN"



## 4. THE REDFUN SYSTEM

### 4.1 SUMMARY OF THE REDFUN REPORT

The REDFUN-report (BEC76) discusses the proposition that a partial evaluator can be a very useful tool for program development. It describes the principles and problems of partial evaluation, shows a number of applications and describes our experience of the experiments done with the REDFUN program.

In this chapter we shall recapitulate from the report what was said about the REDFUN program and its use in the PCDB and GIP/GUP applications and summarize some of the problems which occurred there.

If the contents in the REDFUN-report are familiar this part can be skipped and reading continued at section 4.2. In the part relating to the PCDB application there are some detailed examples showing the use of the partial evaluation technique at program generation.

4.1.1 The REDFUN program (program B in fig 1). The central functions in a program analysis or manipulation program are those which perform the scanning or traversal of the program code. These functions operate in a similar way as the interpreter functions eval, evlis, apply etc. Eval takes as argument a form, evlis a list of forms etc. In REDFUN there is a function redform operation on a form, redargs on an argument-list, and redfun on a functional expression.

The most important function is redform, which takes two arguments, a form and an association list. The form will be partially evaluated with respect to known values of variables, presumably free in the form and appearing on this a-list. If a variable is not present on the a-list or is bound to the special value NOBIN, the variable value is not known.

The NOBIN value is used to conceal previous known values of a variable in other environments.

Redform can be characterized by

```
eval[redform[form,l],ll] = eval[form,append[l,ll]]
```

Examples

```
redform[A,((X . B) (A . Z))] = (QUOTE Z)
```

```
redform[(CAR L),((L . (1 2 3)))] = 1
```

```
redform[(COND ((EQ (CAR X) 'A) (FOO Y)) (T (FIE Y))),
          ((X . (A B C)))] = (FOO Y)
```

Redargs takes a list of forms, normally an argument list and maps over that list and performs redform on each element.

Redfun takes a function expression and a partial binding environment of that function, i.e. a funarg expression.

Example

```
redform [(FUNARG (LAMBDA (X) (CONS X Y)),((Y . A)))] =
          (LAMBDA (X) (CONS X 'A))
```

We return to redform. If its form is the atom NIL, the atom T or a number it is returned unchanged. For other literal atoms a search is made through the a-list and if it is bound the value is returned in a quote-ed expression. See the first example above.

If the form is a list, with a function name as the first element, as

```
(fn arg1 ... argn)
```

different operations will be performed depending on which function class fn belongs to. They are

PURE        for basic functions, without any side-effects, which can be evaluated if all its arguments are known. We call this operation application of function.

Example

```
redform[(CAR L),((L . (X Y Z)))] = (QUOTE X)
```

REDUCER\*    for functions as cond, and, prog etc which have a special argument structure. Such a function has a reducer procedure, stored on its property list under the property REDUCER, which describes how the argument to this function will be processed by REDFUN. The procedure is invoked by

```
apply*[getp[fn,REDUCER],arg1, ...,argn]
```

Example

```
redform[(AND L B (SETQ X A) (FOO Y)),
          ((A . NIL) (B ..T))] =
          (AND L (SETQ X NIL))
```

where B is eliminated because it is always true and will not effect the evaluation of the and-expression and the call to foo will never be reached because the setq-expression will always be false.

OPEN        for functions we want to open, i.e. replacing the function call with the function definition. There are two alternatives. The first is if the arguments have no side-effects and look good enough so that we can do beta-expansion, where the function body,

---

\* In the REDFUN-report this class was called SPECIAL

in which formal arguments have been substituted by the actual and reduced ones, after reduction is inserted at the place for the call.

The second alternative is to perform an open specialization, where the whole lambda-expression is inserted. Reduction inside the expression will also take place.

#### Example

Suppose foo is defined

```
(LAMBDA (X Y) (LIST (CAR X) (CDR Y) X))
```

and is declared open

```
redform[(FOO L (CDR M)),((M . (A B C)))] =  
      (LIST (CAR L) (QUOTE (B C)) L)
```

is an example of beta-expansion.

```
redform [(FOO (PRINT L) (CDR M)),  
          ((M . (A B C)))] =  
      ((LAMBDA (X Y)  
        (LIST (CAR X)  
              (QUOTE (B C)) X))  
       (PRINT L) (QUOTE (B C)))
```

is an open specialization. Although the lambda variable is not needed any more, the actual version did not remove it.

REDUCED \* for functions for which specialization will be performed if some argument or free variable is known. The specialization can be made either open or closed. By closed specialization a new function will be created from the specialized version. This is done if the function is called recursively or if the same specialized code is to be used repeatedly.

---

\* This class was introduced in the GIP/GUP experiment, and included in REDFUN' (program E in fig 1)

**Example**

Suppose foo is defined as

```
(LAMBDA (CHR N)
  (COND ((ZEROP N) (TERPRI))
        (T (PRIN1 CHR) (FOO CHR (SUB1 N))))))
```

A closed specialization through

```
redform [(FOO (QUOTE *) N),NIL]
```

will create a new function foo/1 defined as

```
(LAMBDA (N)
  (COND ((ZEROP N) (TERPRI))
        (T (PRIN1 (QUOTE *)) (FOO/1 (SUB1 N))))))
```

and the value form redform will be the expression

```
(FOO/1 N)
```

**LAMBDA** for functions on which no operations are performed. Their arguments will however be reduced. Functions with side-effects such as put and replace and functions, which depend on side-effects, such as getp fall typically in this group.

Simplification. The resulting expression after partial evaluation needs often to be cleaned up. This is done by simplification rules, called collapsers, such as

(COND X (LIST Y Z))	→	(LIST X Y Z)
(CDR (LIST X Y Z))	→	(LIST Y Z)
(EVAL (QUOTE form))	→	form
(APPLY* (QUOTE CAR) L)	→	(CAR L)
(SET (QUOTE A) 3)	→	(SETQ A 3)

These rules are procedures and are associated with the leading function in the rule. They are stored on the property list of the function under the property COLLAPSER.

Some rules are predefined by the system and new rules can be given by the user. An advise technique inserts new rules in the collapser procedure. If the resulting expression from redform is a list

$$(fn\ arg_1\ arg_2\ \dots\ arg_n)$$

the collapser is invoked by

$$apply[getp[fn,COLLAPSER],arg_1,arg_2,\ \dots,arg_n]$$

Beta-expansion. In beta-expansion a substitution is performed in the body of formal arguments (lambda-variables) to the actual ones. This is made through a substitution package in REDFUN, which traverses the program code similar to the central functions in REDFUN, such as redform etc. There are functions substform, substargs and substfun for which substitution is performed in a form, an argument list and a functional expression respectively. Special action must also be taken here for functions with nonstandard argument structures, such as cond and selectq.

In the first and simplest version the beta-expansion was performed through the following algorithm:

- a. Reduce the arguments to the function, which will be opened.
- b. Perform substitution in the function body of lambda-variables to these arguments, reduced in step a.
- c. Call redform with the new function body.

This algorithm has some disadvantages. We are wasting time if we perform substitution on parts in the program code, which will later be eliminated. The arguments already reduced, which have now been inserted in the code will be repeatedly reduced.

An improvement was made in a later version. When encountering a cond-expression (and selectq-expression) calls to redform were performed on the predicates in an attempt to throw away the then-clause if the predicate was reduced to NIL.

Other improvements. Another place where inefficiencies occurred was when the collapsers had simplified an expression. A recursive call was then made to reform to try to reduce it further.

This was necessary in cases like

```
(APPLY* (QUOTE CAR) (QUOTE (A B C)))
```

for which the apply\*-collapser will return

```
(CAR (QUOTE (A B C)))
```

which of course can be reduced further. But in cases like

```
(CONS X (LIST Y Z))
```

which collapses to

```
(LIST X Y Z)
```

no further reduction is needed.

This was solved in one application, by letting the recursive call to redform after a collapsing be taken away and the value from the collapser remain as it is. This means that sometimes we shall receive expressions which are not satisfactorily simplified. By looping on the top level of the form

```
fn:=redfun[fn,al]
```

until fn converges this problem is solved. This seems to be wasteful but in some cases an improvement was made.

To speed up some conditionals a modified cond was introduced on the form

```
(CONDVAR (v1...vm) (p1 e1)...(T en))
```

where all  $v_i$ 's are those variables, which can occur in any of the  $p_i$ . The condvar reducer could then check if all  $v_i$  had known values and in that case perform an eval instead of redform on every  $p_i$ .

#### 4.1.2 REDFUN in the PCDB application (application C1 in fig 1).

The PCDB system is described in (SAN71, HAR74, HAR73). It maintains a data base of formulas in predicate calculus and works as a program generator in order to create efficient LISP procedures for storage and retrieval of such formulas. Axioms in the predicate calculus are also compiled to LISP code. The REDFUN-rapport (BEC76) discusses the principles of the design considerations taken to perform the program generation in this application.

REDFUN was primarily designed for this application but it was found that specializing of code was more general and could be used in other applications, so instead of including REDFUN in the PCDB system a separate package was made.

4.1.2.1 Generation of stordef-procedures. Every relation in PCDB will have a number of procedures related to it, one for storage of an assertion with the relation, one for explicit retrieval and some others. The code generated for such a procedure is driven by a number of parameters, either given by the user or implicit calculated by the system. These parameters describe different properties of the relation, such as number of arguments, datatype of the arguments and the "one-manyness" between the arguments. The method used to generate these procedures can be described as

- a. The different relations can be grouped together and for each group a general procedure was written for storage of a relation in that group. The same was done for the retrieval procedure and the other procedures. The parameters correspond to free variables in these procedures.



- b. In these procedures calls were often made to a library of auxiliary procedures. Some of them could also in their turn be opened and some formed a set of run time procedures, mostly for accessing the property lists (variations of put, getp, addprop etc).
- c. When a procedure was needed for a relation a funarg-expression was created by the general procedure with the free variables bound to the parameter values and transferred as funarg-variables.
- d. The funarg-expression was then given to redfun and the result was then a specialized version suitable for that relation.

Some examples may clarify this.

Suppose we give to PCDB:

```
RELATION (CHILD 2 (AA AA) (ONE MANY))
```

We will then define a relation child of two arguments, both of datatype literal atom (AA AA). The "one-manyness" declaration (ONE MANY) says that if we have

child(x,y) with meaning "y is a child of x"

then for each x there can be several y, but for a given y there can only be one x.

```
RELATION (MARRIED 2 (AA AA) (ONE ONE))
```

defines married as a one-to-one relation.

```
RELATION (AGE 2 (AA SX) (ONE MANY))
```

defines age, whose second argument is an arbitrary S-expression, in this case used for an integer. Another storage convention must then be used. Normally if an argument is a literal atom, the argument's property lists are used to store the assertion, otherwise the relation's property list is used.

We can then with PCDB store assertions, such as

```
STORE (CHILD ANDERS KARIN)
STORE (MARRIED ANDERS EVA)
STORE (AGE ANDERS 30)
```

and perform retrievals, such as

```
FETCH (AGE ANDERS)
TEST (MARRIED OLLE EVA)
```

Let us follow how the storage procedures for these relations are generated. These examples will demonstrate the complexity of the code this version of REDFUN can operate on, but also that the reduction, when functions on several levels are beta-expanded are non-trivial to follow. These examples are shown again in appendix I, where the beta-expansion is done one level at a time to make it easier for the reader to follow the reductions.

All relations belong to the class cleanlink. For each relation a funarg-expression is created containing the general storage procedure for relations in that class. The funarg-expression is

```
(FUNARG
  <LAMBDA (A B)
    (COND
      <(ONEONE ONE)
      (PROG (ROOT)
        (RETURN (COND
          ((TESTER P A B LOC
            (CAR TYP)
            (CADR TYP))
          (ROOT)
          ((STOREP (REV P)
            B A (QUOTE ONE)
            LOC
            (CADR TYP))
          (RPLACA ROOT B)
          (T (APPLY* (FILLAND ONE)
            (QUOTE (STOREP P A B (CADR ONE)
              LOC
              (CAR TYP)))
            (QUOTE (STOREP (REV P)
              B A (CAR ONE)
              LOC
              (CADR TYP))
            (R ONE LOC TYP))
```

The variables r, one, loc and typ are transferred as funarg-variables. These variables correspond to the parameters given by the user or calculated by default by the system. They are bound to values in the funarg-block (only shown here by a pointer). The fourth element in the funarg-expression is used to inform redfun what variables in the funarg-block will form the a-list of variables with known values in REDFUN.

The function rev returns the reversed relation and is defaulted to concatenate REV before the relation name

```
rev[CHILD] = REVCHILD
```

The functions tester, storer, getter and comparer are declared open and will be beta-expanded and the functions oneone, filland and fillfunc are pure. Their definitions are

TESTER

-----

```
<LAMBDA (R A B LOC TA TB)
  (AND (CAR (SETQ ROOT (GETTER R A LOC TA)))
    (PROG2 (SETQ ROOT
      (APPLY* (COMPARER (QUOTE ONE)
        TB)
      (CAR ROOT)
      B))
    T>
```

STCRFR

-----

```
<LAMBDA (R A B N L TI)
  (APPLY* (FILLFUNC N)
    (GETTER R A L TI)
    B>
```

GETTER

```

<LAMBDA (R A L TI)
  (OLDCOND
    ((AND (MEMB (QUOTE AA)
                 TI)
          (AA A))
      (SELECTQ L
        (ARGS (GETROOT A R))
        (PRED (RGETROOT (GETROOT
                          R
                          (QUOTE TRUEFOR))
                     A))
              ((HF CYCYCHF)
               (GETROOT A R))
              NIL))
    ((AND (MEMB (QUOTE HX)
                 TI)
          (HX A))
      (SELECTQ L
        (ARGS (GETROOT (CAR A)
                        R))
        (PRED (RGETROOT (GETROOT
                          R
                          (QUOTE TRUEFOR))
                     (CAR A)))
              ((HF CYCYCHF)
               (GETROOT (CAR A)
                        R))
              NIL))
    ((MEMB (QUOTE SX)
           TI)
      (RGETROOT (GETROOT R (QUOTE TRUEFOR))
                 A>

```

COMPARER

```

<LAMBDA (N TI)
  (COND
    ((MEMB (QUOTE SX)
           TI)
      (SELECTQ N
        (MANY (FUNCTION MEMBER))
        (ONE (FUNCTION EQUAL))
        NIL))
    (T (SELECTQ N
      (ONE (FUNCTION EQ))
      (MANY (FUNCTION MEMB))
      NIL>

```

ONFONE

```

<LAMBDA (ONE)
  (EQUAL ONE (QUOTE (ONE ONE)>

```

FILLAND

```

<LAMBDA (ONE)
  (COND
    ((EQUAL ONE (QUOTE (ONE MANY)))
      (FUNCTION REVAND))
    (T (FUNCTION AND>

```

FILLFUNC

```

<LAMBDA (NC)
  (SELECTQ NC
    (ONE (FUNCTION FILLROOT))
    (MANY (FUNCTION ADDRROOT))
    NIL>

```

The functions getroot and rgetroot in getter are functions which belong to the set of run time procedures. They are declared to belong to the function class lambda, and getroot for example is defined such that

```
car[getroot[a,p]] = getp[a,p]
```

and oldcond is a variant of cond, which assumes the last predicate in a cond always to be true and can therefore be substituted into T by redfun. The function filland returns either and or revand, where the function revand is as and but processes its arguments backwards.

The funarg-block for the child-relation corresponds to the association list

```
((R . CHILD) (ONE . (ONE MANY)) (LOC . ARGS) (TYP . (AA) (AA))))
```

The code for child's stordef will after reduction be

```

<LAMBDA (A B)
  (AND (FILLROOT (GETROOT B (QUOTE REVCHILD))
    A)
    (ADDRROOT (GETROOT A (QUOTE CHILD))
    B>

```

In the code fillroot and addroot corresponds to put and addprop resp. The reversed relation to father is by default revfather and they are both used as property names

A collapser rule

```
(REVAND X Y) → (AND Y X)
```

has been used.

The stordef procedure for married will be

```
<LAMBDA (A B)
  (PROG (ROOT)
    (RETURN
      (COND
        ((AND <CAR (SETQ ROOT
          (GETROOT A (QUOTE MARRIED)
        (PROG2 (SETQ ROOT
          (EQ (CAR ROOT)
            B))
          T))
        ROOT)
        ((FILLROOT (GETROOT B (QUOTE REVMARRIED)
          )
          A)
        (RPLACA ROOT B>
```

and for age

```
<LAMBDA (A B)
  (AND (FILLROOT (RGETROOT (GETROOT (QUOTE REVAGE)
    (QUOTE TRUEFOR))
    B)
    A)
  (ADDROOT (GETROOT A (QUOTE AGE))
    B>
```

The code generated for these stordef-procedures will not store any assertion which contradicts the "one-manyness" declaration. If it contradicts, a NIL value is returned.

4.1.2.2 Example from the axiom compilation step. An axiom in the predicate calculus is given to the PCDB system as a clause in an implication form such as

$$R(y,z) \wedge S(x,y) \wedge P(x) \supset T(x,z) \quad (*)$$

$$R(o,w(c,a),s) \supset S(g(c,o,a),s)$$

The axiom is converted to a deterministic procedural form after a schemata described in Sandewall (SAN73). The code is then generated after this form.

An axiom can be declared to be used in one or more different modes, such as to answer closed questions (YES/NO questions) or open questions (a question returning objects from the data base). In the first case, if the axiom is to be used for forward deduction (at storage time) or backward deduction (at retrieval time) and finally if the deduction shall be performed in a breadth-first or depth-first order.

Suppose that the first axiom (\*) is to be used as a backward axiom, used in a depth-first manner, and to be able to answer open questions, i.e. of the type "for a given x find all z which satisfy T(x,z)". The axiom can be expressed in the procedural form something like

```
For a given x find all z such that
P(x) holds and for each y such that
S(x,y) holds create a new subquestion
    "find all z which satisfy R(y,z)"
```

The steps in generation the code are as follows:

- a. For each literal, except the first one, create a pair of expressions containing a call to the functions bind and cont respectively, which are two general functions for accessing the data base and for maintaining the result(s) from the access. The arguments to these two functions are quote-ed expressions and accordingly known.

- b. A prog-expression will be created by these pairs and by an expression, which controls how the subquestion will be treated.
- c. The two functions bind and cont are declared open to REDFUN, so the whole prog-expression can be given to redform, which performs beta-expression and partial evaluation on it. Some optimization of the goto structure will also be done.
- d. This code will be inserted in a procedure associated with the triggering relation, in this case the relation r.

Axioms compiled in other modes will be treated in similar ways. Axioms containing predicate calculus functions, as in the second example, will be taken care of in a special way in a pre-step before the compilation. This is not discussed here.

Finally we shall show the above steps, running the example through PCDB and REDFUN.



The first two steps will produce

```

<LAMBDA (N* X)
  (PROG (Y YQUE)
    (AXNAME AX0)
    M1 (BIND NIL (QUOTE (P X))
        (QUOTE B0)
        (QUOTE M2))
    B1 (CONT NIL (QUOTE (P X))
        (QUOTE B0)
        (QUOTE M2))
    M2 (BIND (QUOTE (Y))
        (QUOTE (S X Y))
        (QUOTE B1)
        (QUOTE M3))
    B2 (CONT (QUOTE (Y))
        (QUOTE (S X Y))
        (QUOTE B1)
        (QUOTE M3))
    M3 (AND (APPLY* (QUOTE AUXRECFIND)
                  (LIST (QUOTE P)
                        Y))
          (RETURN T))
    (GO B2)
    B0 (RETURN>

```

Bind and cont have as arguments a variable list, the literal and two labels. For an open question the variable list contains those variables from the literal for which values are retrieved and for a closed question it is NIL. The labels are used to go to if the retrieval was succesful or not.

The code will after reduction be

```

<LAMBDA (N* X)
  (PROG (Y YQUE)
    (AXNAME AX2)
    M1 (AND (SYSTEST (LIST (QUOTE P)
                          X))
          (GO M2))
    (GO B0)
    M2 (SETQ YQUE (SYSFETCH (LIST (QUOTE S)
                                X)))
    B2 (COND
        (YQUE (SETQ Y (CAP YQUE))
            (SETQ YQUE (COR YQUE))
            (GO M3))
        (T (GO B0)))
    M3 (AND (AUXRECFIND (LIST (QUOTE P)
                              Y))
          (RETURN T))
    (GO B2)
    B0 (RETURN>

```

In this code we still have calls to systest and sysfetch, which are two internal functions, where systest is used to test if something is stored in the data base, in this example if  $P(x)$  is stored, and sysfetch makes a retrieval, in this case binding to yque the result given by  $S(x,?)$ . These functions can be further opened, but they are useful to have in the code if one wants to trace all accesses in the data base.

Systest is essentially defined as

```
(LAMBDA (U)
  (GETP (CAR U)
    (QUOTE TESTDEF)))
```

and sysfetch is defined similarly.

If they are declared open the final code will be

```
<LAMBDA (N* X)
  (PROG (Y YQUE)
    (AXNAME AX2)
    M1 (AND (EQ (GETP X (QUOTE P))
               (QUOTE TRUE))
          (GO M2))
    (GO B0)
    M2 (SETQ YQUE (GETP X (QUOTE S)))
    B2 (COND
        (YQUE (SETQ Y (CAR YQUE))
              (SETQ YQUE (CDR YQUE))
              (GO M3))
        (T (GO B0)))
    M3 (AND (AUXPECFIND (LIST (QUOTE R)
                              Y))
            (RETURN T))
    (GO B2)
    B0 (RETURN>
```

#### 4.1.3 REDFUN in the GIP/GUP experiment (application C2 in fig 1).

In this experiment we wanted to test REDFUN on a program, which was not particularly designed to suit or be influenced by REDFUN. The GIP/GUP package was developed by Ö. Oskarsson (OSK73) and was intended for parametrized input and output of property lists. The parameters described what data to store or retrieve and for example what layout the user wants his data to be printed in, such as indentation, special characters to separate atoms etc. There are totally 28 parameters to use, almost all of them have default values attached to them.

This program became of course very flexible but in some applications, where the program was used several times with the same set of parameter values, it could be inefficient. The experiment was to try to apply REDFUN on GUP (the General Output Program) in order to partially evaluate it. The aim of this experiment was to see what problems occur, when REDFUN is used on a program written without any restrictions in mind, and to see what extensions should be made to REDFUN so that it can manipulate a larger class of LISP program.

One extension that was necessary from the beginning was to introduce the REDUCED option, i.e. closed specialization. The call structure between the functions in GUP was such that the same auxiliary functions were called from different places. Beta-expansion could not be performed because of side-effects and open specialization was not appropriate because of recursion.

An example taken from the REDFUN report (BEC76) shows the complexity of a function, which was reduced.

XTPR is declared REDUCED, REALIND and REALPROP are declared OPEN and TAKEIN, TAKENDL and TAKEFL are declared PURE. XTPR/F1 is the new version of XTPR generated by REDFUN. The function OUTIND has been reduced to OUTIND/F1, but is not shown here. It uses however IDTI, IDTP and CARRTERM as free variables, which are bound in XTPR/F1.

Before:

```

(REALIND
  <LAMBDA (I)
    <COND
      ((GETL ILI (QUOTE INTNAME)
        (T I))

(REALPROP
  <LAMBDA (I)
    <COND
      ((TAKEFL FLI (QUOTE FUNC))
        (APPLY* I A))
      (T (GETP A I))

(XTPR
  <LAMBDA (IX)
    (PROG (P ILI FLI CARRTERM I SAVE PUTFUNC IDTI
            IDTP)
      (SETQ ILI (COP IX))
      (SETQ IX (CAR IX))
      <COND
        ((NULL (ATOM (CAR ILI)))
          (SETQ FLI (CAP ILI))
          (SETQ ILI (COP ILI)
        (SETQ PUTFUNC (TAKEIN ILI (QUOTE OUTFUNC)))
        (SETQ I (REALINE IX))
        (SETQ P (REALPROP I))
        <COND
          (OUTFUNC (SETQ P (APPLY* OUTFUNC P)
        (COND
          ((NULL P)
            (RETURN)))
        (SETQ IDTI (PLUS (TAKEIN ILI (QUOTE INDENT))
          )
          IDTC))
        (SETQ IDTP (PLUS (TAKEIN ILI (QUOTE INDENTP)
          )
          IDTI))
        (SETQ CARRTERM (TAKENDL ILI (QUOTE CARRTERM)
          ))
        (OUTIND I P (TAKEFL FLI (QUOTE VERT)))
        <COND
          ((SETQ SAVE (TAKEIN ILI (QUOTE SAVE)))
            (PLACA SAVE (APPEND (CAR SAVE)
              (DTST P)
            (RETURN>))

```

After:

```
(XTPR/F1
  <LAMRDA NIL
    (PROG (P CARPTERM IDTI IDTP)
      (SETQ P (GETP A (QUOTE RELATION)))
      (COND
        ((NULL P)
          (RETURN)))
      (SETQ IDTI 2)
      (SETQ IDTP 4)
      (SETQ CARPTERM (QUOTE =))
      (OUTIND/F1 NIL P)
      (RETURN>)
```

Remark. Notice that there are no goto's in the XTPR-function.  
 This REDFUN-version (step E in fig 1) could not handle  
 assignments properly when goto's were involved.

4.1.4 Conclusions from the usage of the REDFUN program (step D in fig 1). Here we shall list some of the problems, which occurred during usage of REDFUN in these applications and experiments and propose a number of extensions to a revised version, which we shall call REDFUN-2.

4.1.4.1 Experiences from the PCDB-application. The REDFUN program was from the beginning designed for the specific needs found in PCDB. The hierarchical structure of the auxiliary functions made the beta-expansion straight forward to perform and the restricted use of prog's made the analyzing of prog-statements fairly simple. Some minor problems which occurred can however be mentioned.

During the beta-expansion a variable in an auxiliary function was substituted more than once by an expression, which made a property list access. During execution this access was made unsatisfactorily several times. This did not introduce any incorrectness, but efficiency suffered. An example occurred in a function defined as

```
[LAMBDA (R B)
  (COND
    ((NULL (CAR R))
     (RFLACA R B))
    ((EQ (CAR R)
         B)
     T)
    (T NIL]))
```

and the lambda-variable r was bound at opening to

```
(GETROOT S A)
```

where getroot was defined earlier in section 4.1.2.1.

The resulting code

```
(COND
  ((NULL (CAR (GETROOT S A)))
   (RPLACA (GETROOT S A)
            B))
  ((EQ (CAR (GETROOT S A))
       B)
   T)
  (T NIL))
```

was not satisfactory. The problem was temporarily solved in PCDB by rewriting the few functions in which this occurred, to prog's and to a prog-variable binding the value after the property list access, such as

```
(LAMBDA (R B)
  (PROG ((ROOT R))
    (RETURN (COND
      ((NULL (CAR ROOT))
       (RPLACA ROOT B))
      ((EQ (CAR ROOT)
           B)
       T)
      (T NIL))
```

Another problem occurred when the bind function was extended also to take an optional argument indication what access function should be used. Instead of sysfetch, which finds explicit stored facts, one can use for example sysrecfind also to find facts implicitly stored (see the example in section 4.1.2.2). The bind-expression will then appear as

```
(BIND (QUOTE (Y))
      (QUOTE (S X Y))
      (QUOTE B1)
      (QUOTE M3)
      (QUOTE SYSRECFIND))
```

Bind was defined approximately as

```

[CLAMBDA (QVARS LIT B M SYSFN DEPTH)
  [COND
    ((NULL SYSFN)
      (SETQ SYSFN (SYSFNDFLT QVARS)
        ...])]

```

and if a function is not supplied, one is calculated by default. REDFUN however could not handle this case, when a variable, which was known, received a new value. The bind function was rewritten to

```

[CLAMBDA (QVARS LIT B M SYSFN DEPTH)
  (BINDAUX QVARS LIT B M (BINDSYSFN SYSFN QVARS)
    DEPTH)]

```

and bindsysfn as

```

[CLAMBDA (SYSFN QVARS)
  (COND
    ((NULL SYSFN)
      (SYSFNDFLT QVARS))
    (T SYSFN))]

```

and bindaux as the old bind, in which the cond-expression was deleted. Bind and bindaux are declared open and bindsysfn is declared pure. Instead of rebinding a known variable by a setq it is rebound through a function call, which REDFUN could handle.

The collapser were in some cases not generally written, but restricted to use in PCDB. If however PCDB is changed or extended, cases may easily occur which will be erroneously transformed by these collapser. Such an example is the collapser for apply\*, which performs the following transformations

```

(APPLY* 'CAR '(A B C))  +      (CAR '(A B C))
(APPLY* 'CAR L)          +      (CAR L)
(APPLY* 'AND 'X 'Y)      +      (AND X Y)

```

but will transform incorrectly

```

(APPLY* 'AND X Y) to (AND X Y)

```



There is no way to escape from a collapse without doing a transformation and if we return with the same expression, the collapse will recursively be called again and we enter an infinite loop.

4.1.4.2 Experiences from the GIP/GUP experiment. In the GIP/GUP experiment we expected more problems, which would probably force us to make a redesign of REDFUN. The reduced option was introduced directly in order to make closed specialization.

One main problem was assignment of variables. It is natural for a programmer to use assignments, particular in prog-expressions. There were cases like

```
(LAMBDA (X Y)
  (PROG (P1 P2 P3)
    (SETQ P1 (CAR X))
    (SETQ P2 (CADR X))
    (SETQ P3 (CADDR X))
    ...
  ))
```

in which x was known through the call and then naturally both P<sub>1</sub>, P<sub>2</sub> and P<sub>3</sub> would be known.

A more difficult problem to solve was when the assignment was inside a conditional expression as in

```
(COND
  ((EQ X 'A) (SETQ Z (CAR L)) (FOO Z))
  (T (FIE Z)))
```

If we assume that l has a known value then z will be known in the true-clause after the assignment, but we know nothing about z in the false-clause and in statements following this cond-expression. When conditionals appear within each other and assignments are done on different levels the problem of keeping track of them will become rather complicated.

The first version of REDFUN could not take care of these assignments, but in the experiment the problem was solved by introducing free variables which could notify reducers that assignments had been performed. This solution was of a more temporary nature and a more systematic design of this problem was desirable.

Prog-expressions have only been reduced in a restricted way. In principle every statement will be reduced by redform and a simple clean-up of go's and labels is done. When assignments occur in a prog, which happened in the GIP/GUP case, they could only be taken care of when they occurred before the first label. After that label no assignments, even if they were assigned to constant values, were taken care of.

At beta-expansion it has already been discussed that efficiency problems can occur both in the substitution phase - substitutions are done in parts, which will later be eliminated - and in the following reduction in the body - the arguments, which have already been reduced will be overhauled again by redform. Another problem in REDFUN is that if the user introduces his own functions, which have a special argument structure, he must insert code in substform to show how substitution had to be performed in the arguments.



4.1.5 The REDCOMPILE program. This program will not be included in this work, but will be referenced in sections 7 and 8. Here follows a very short description of the ideas in that program. The purpose with REDCOMPILE was to speed up the reduction process. It is useful when REDFUN operates on a program P several times with the same set of variables with known values every time. REDCOMPILE can then generate a generator version of P, which in its turn is used to generate the specialized version.

Let us follow an example. Suppose foo is defined as

```
[LAMBDA (X Y Z)
  (COND
    (X (COND
        (Y (CONS X Z))
        (T Z)))
    (T (LIST Y Z]))
```

If we assume x to be a variable with known value, REDCOMPILE can transform this code into a program foo'

```
[LAMBDA ....
  (COND
    [(GETVAL X)
     (LIST (QUOTE COND)
           (LIST (GETARG Y)
                 (LIST (QUOTE CONS)
                       (GETARG X)
                       (GETARG Z)))
           (LIST T (GETARG Z))
          (T (LIST (QUOTE LIST)
                  (GETARG Y)
                  (GETARG Z)))]
```

When executed, this program will generate the appropriate specialized version of foo.

Exactly how the lambda-variables are bound is left aside here, but getval and getarg retrieves the value resp. the form a variable stands for.

The first cond can be evaluated at generation time, but the second one is postponed to execution time (of the generated program).

At opening of

```
(FOO 'A (CAR L) (FIE L))
```

foo' will directly generate

```
(COND ((CAR L) (CONS 'A (FIE L)))
      (T (FIE L)))
```

The actual version of REDCOMPILE has been used in PCDB and all functions there, which are defined as open, have been run through REDCOMPILE. Still a lot of problems remain to solve in order to get a REDCOMPILE system, which can process more complex code. Some problems are more difficult to solve in REDCOMPILE than in REDFUN, and have to be solved either by running REDFUN as a post-processor, or by instructing REDCOMPILE to insert calls to redform into the code it generates.

#### 4.2 A STUDY OF MANIPULATION IN THE WTFIX FILE (STEP C3).

The INTERLISP system implemented on the IBM 360/370 was developed in such a way that a large portion of the LISP code could be directly transferred from the INTERLISP-10 system when some minor changes had been performed. Some features in INTERLISP-10 have been excluded in the basic INTERLISP 360/370 system, for example history, CLISP and DWIM. Some of these packages have been implemented locally in the INTERLISP 360/370 system. This work was mainly done by Jim Goodwin. One problem with these packages was that they were spread over many file. When the history package was implemented, one task was to extract from the file WTFIX those parts which were used only by that package, omitting those parts which were included there to serve the packages DWIM and CLISP. This extraction was done manually, by scanning down the code in the file and marking what parts could be deleted. This task is not so trivial as it perhaps sounds. One must know the functions and variables which are only used by these "unwanted" packages and find all special cases in code which are there only to serve them. When this task is made manually the comments in the texts gives of course a lot of valuable information.

Our question was then, what should an automatic system look like in order to help us in this case. A discussion with Jim Goodwin resulted in a list of all changes done in the file. We then analyzed this list to see if any of our program manipulation programs, especially REDFUN, could perform such changes. What we think we need is a more intelligent editor, which not only knows about the syntax, but also more about the semantics of the language, and therefore is able to perform more advanced changes in the code.

No such "intelligent editor" was written, but the detailed study of the problems resulted in some requirements on a partial evaluator. We shall here discuss the example from that point of view.

Changes of a partial evaluation character. A number of flags in the source code, eg dwimflg and clispflg specified explicitly parts of the code that could be removed. These flags could be assumed to have the value NIL and the reduction is then a typical task for REDFUN to perform. There were also some calls to functions, which should be deleted, and for which we could assume a constant value to be returned. The reduction could then be performed like for variables.

There was also a case where a variable was three valued and one value could be excluded. Parts in the code where the variable had this excluded value, could be deleted. Partial evaluation with respect of the set of permissible values for a variable, or the information that a variable can not have certain values, can not be done by REDFUN, and was selected as one requirement on the next generation program.

In another case we had

```
(SELECTQ (CAR FAULTX)
  .. a number of cases ,,
)
```

and a number of these cases had to be removed because no special treatment should be performed in these cases. This task could be seen as a partial evaluation task if we could say something like "In the lists that are possible values of faultx, the first element can not be any of  $a_1$ ,  $a_2$  etc", or perhaps more correctly, "if the first element is one of  $a_1$ ,  $a_2$  etc we are not interested in special treatments of such lists at any place in the code". This case can not either be done by REDFUN.

Changes where partial evaluation is not sufficient. A number of variables, mostly prog-variables, and calls to certain functions had to be deleted entirely from the code. This deletion can not be performed as a partial evaluation task, because we can not assume any value(s) for these variables and function calls. Special care must also be taken when they are deleted from the code. The question is if any of the environment, in which the variable or call is placed, also had to be deleted.

In the example

```
(COND ((LISTP X) (CAR Z))
      (T (FUM X)))
```

where z is assumed to be deleted, the whole car-expression can be removed, but the user had to tell if he wants to keep the listp-check or not. We can either get

```
(COND ((LISTP X))
      (T (FUM)))
```

or

```
(FUM X)
```

Another example. In WTFIX there was a variable fixclk, which had to be deleted. The assignment

```
(SETQ FIXCLK (CLOCK 2))
```

has therefore no sense, but it is not obvious if the third form in the expressions also should be deleted. If the assignment is performed in a context, where the value of the assignment is of interest, it can not simply be deleted, and also if it performs side-effects. In other cases the expression can be taken away, eg if the assignment is performed as a statement in a prog.

Transformations. To clean up in the code we wanted to perform a number of transformations, such as

```
(SELECTQ (CAR FAULTX)
  (T ***)
  NIL)
```

to

```
(COND ((EQ (CAR FAULTX) T) ***))
```

Most of them could be performed through collapse rules in REDFUN.

Still of course there is a number of cases which are really difficult to perform automatically. They involve a rather deep understanding and knowledge of the whole INTERLISP-system and can only be carried out manually.

4.2.1 Conclusions from the study. Partial evaluation and the other operations in REDFUN could perform some of the tasks found in this study. New requirements on the next version of REDFUN was to extend the ability to handle more information about the values a variable can be bound to. In this study we could use the knowledge that a variable could only have two different values. Another requirement was to handle information about list structures, such as the case when we knew that the first element in a list only could be certain objects. In the next version REDFUN-2 the first requirement will be included. The second one is much more complex to realize in a general way and will therefore be excluded.

In other discussions about requirements of the REDFUN system there is a need to keep track of the datatype(s) of the values a variable can be bound to. Redundant datatype checks can then be removed from the code. This will also be included in REDFUN-2.



## 5. DESIGN ALTERATIONS FOR REDFUN-2

In this chapter we will first list the new features which will be included in the new version of the REDFUN program. This new version is called REDFUN-2 to distinguish from the old version. The main design considerations in this implementation are described in the second section. These two parts correspond to step F in the design iteration scheme shown in chapter 3. The actual implementation of REDFUN-2 will be described in chapter 6.

### 5.1 NEW FEATURES IN REDFUN-2.

The experiments with REDFUN, reported in the former section, and other desirable features led us to propose the following main extensions:

5.1.1 Extended range of information about variable values. In REDFUN we could only make use of the fact that a variable had a constant value. We would like to extend this also to make use of the fact that we at a specific point in the program know

- all the different values a variable can be bound to
- values a variable cannot be bound to
- the datatype(s) of the values a variable can be bound to

Suppose that X has the value A

Y has one of the values B, C or D

Z cannot have the value A

V is an integer

We can then perform the following reductions:

```
(COND ((EQ X 'A) (FOO X))
      (T (FIE X)))      → (FOO A)

(COND ((EQ Y 'A) (FOO X))
      (T (FIE X)))      → (FIE X)

(COND ((EQ Z 'A) (FOO X))
      (T (FIE X)))      → (FIE X)

(COND ((LISTP V) (FOO X))
      (T (FIE X)))      → (FIE X)
```

This extension is a natural next step and the need of it has been found both in the WTFIX-study (4.2) and in other discussions of the usefulness of the REDFUN system.

Our system will not be able to manage logical expressions in order to describe the values a variable can be bound to, such as

"the variable x has the value 5 if either y is 2 or z is 6"

The system would have needed a more advanced deduction capability in order to handle such expressions and we decided not to include it in this version.

5.1.2 Handling of values from expressions. The extended range of information about variable values will make it necessary for the system to be able to keep track of value information about arbitrary expressions in the code. If x can have either 1 or 2 as value the expression `add1[x]` will consequently have the values 2 or 3. Every branch in a cond-expression can have known values and it is then possible to derive all the values the cond-expression can return. The expression

```
(COND ((EQ X 'A) 10)
      ((EQ X 'B) 3)
      (T 5))
```

will naturally return either 3,5 or 10 as value. A number of functions will always return a value of a certain datatype, which in many cases can be useful to know.

For some functions the value can be derived even if we only have partial knowledge about its arguments. For the expression

```
(PUT A B 'VALUE)
```

we know from put's definition that its third argument will always be returned as value. Another case is if we know in an eq-check that its first argument can only have A or B as values and its second argument cannot have those values we can conclude that the eq-check will always be false. We call this a pseudo-computation, when the value can be calculated without really evaluating the expression.

5.1.3 Systematic handling of side-effects, especially assignments. All side-effects, such as assignments, destructive changes in list structures, input/output etc must be detected and carefully processed. A form performing side-effects can normally not be deleted from the code. Rearrangements of such forms can only be done in restricted cases. If side-effects interact with each other or with other code they must of course occur in the same order in the rearranged code.

In side-effects appear in an argument to a pure-function and the value of this argument is known, we shall want to apply the function to its arguments. The form with the side-effect must remain in the resulting expression. Suppose we have

```
(CONS (PUT A B 'C) '(D E))
```

and if cons is pure the form could be reduced to

```
(PROGN (PUT A B 'C) (QUOTE (C D E)))
```

In every form an analysis must be performed to find those variables which can be assigned there, but also to distinguish between variables, which will always be assigned and those which will only be assigned in some branch of the expression.

Here we shall show some different cases which the system must be able to handle. Suppose x has the value A and z one of the values NIL or B.

a. (PROGN (CONS (SETQ X Z) (FOO X)) (FIE X))

The assignment of x will have the result that x both at the call to foo and fie will have one of the values NIL or B.

b. (PROGN (AND Z (SETQ X Z) (FOO X)) (FIE X))

The assignment here will have the result that at the call to foo, x will have the value B. If z has the value NIL the evaluation in the and-expression will be ended and the assignment never performed. At the call to fie we cannot be sure if the assignment has been done and can only deduce here that x will have one of the values NIL, A or B.

c. (PROGN (AND (SETQ X Z) (FOO X)) (FIE X))

In this case, when the assignment is done first in the and-expression it will always be performed. At the call to fie we can exclude the old value A for x, otherwise it works as in the case b.

In cond-expressions an analysis must also be performed to check if a variable will always be assigned there regardless of the path taken in the expression.

To take care of assignments in a proper and systematic way is probably the most important extension compared to the basic REDFUN version (program B in fig 1). In the PCDB application (4.1.2) there were no assignments in the code which was processed by REDFUN, but in the GIP/GUP-program (4.1.3) the assignments caused a lot of problems for REDFUN, although some of these problems could be solved temporarily in the REDFUN'-version (program E in fig 1). If one wants REDFUN to process ordinary written code one cannot expect such code be written without assignments.

5.1.4 Extraction of variable properties from the code. In conditionals the predicate often gives information about variables, which can be used during the processing in the different branches. If we have

```
(COND ((EQ X 'A) (FOO X))
      (T (FIE X)))
```

and nothing is known about x before the cond-expression, we can in the true branch derive that x has the value A and in the false branch that it cannot have A as value.

The system shall also be capable of deriving datatype information about values a variable can be bound to, such as

```
(COND ((LITATOM X) (F1 X))
      (T (F2 X))))
```

Here we know that x is a literal atom at the call to f1 and that x cannot be a literal atom at the call to f2.

It is clear that a program contains masses of information which could be extracted but we must here restrict ourselves to extracting only that information about a variable which can be obtained reasonably economically.

In our system information will be derived for pure LISP predicates, such as eq, memb, litatom etc. and for logical functions and conditional expressions. We restrict the extraction only to be done if the information about a variable can be described in one of the three ways mentioned in 5.1.1, and accordingly to that not when a logical expression is needed to describe the information.

In the expression

```
(COND ((OR (EQ X 'A) (EQ X 'B)) (FOO X))
      (T (FIE X)))
```

we can derive that x is either A or B in the true branch and that x cannot be A or B in the false branch. In the expression,

```
(COND ((OR (EQ X 'A) (EQ Y 'B)) (FOO X))
      (T (FIE X)))
```

however, we can only derive information about the variables, x and y in the false branch, i.e. that x cannot have the value A and y not the value B. In the true branch we cannot know which of the two forms in the or-expressions was calculated to true. The information of x and y in the true branch can only be described by a logical expression and will not be derived.

5.1.5 Reduction of prog-expression. REDFUN could only handle restricted structures of prog-expressions. In REDFUN' (program E in fig 1) assignments of variables were only taken care of before the first label and after that label no information about the variables was assumed to be known. In arbitrary loop-structures it is a complicated task to perform a perfect analysis of assignments and how these assignments will affect the reduction. In REDFUN-2 an analysis will be performed to find the goto-structure and to detect loops, and those variables which may be assigned in these loops. A variable assigned in a loop is often treated as to have an unknown value at the reduction in the loop. In some cases, if the variable is assigned to constant values or values of certain datatype, such information can be used to avoid that the variable is assumed to have unknown values at reduction of the loop, such as in the example

```

(PROG (Y)
  (SETQ Y 0)
  LOP
  (COND ((FOO X Y) (GO OUT)))
  (OR X (SETQ Y 1))
  (SETQ X (FIE X Y))
  (GO LOP)
  OUT
  .
  .
  .)

```

Here we know that the variable y only can have the values 0 and 1 in the loop (if we assume foo and fie not to perform any side-effects of y).

This analysis of side-effects has to be performed before the reduction. Therefore assignments in parts which should be eliminated may sometimes affect the knowledge we have about a variable before the reduction of the loop. A small example illustrates such a case which we assume the new system not to be able to reduce completely.

```

(PROG (X Y)
  (SETQ X 2)
  LOP
  (COND ((FOO X) (GO OUT)))
  (AND (EQ X 1) (SETQ Y 3))
  (SETQ X (ADD1 X))
  (AND (EQ Y 3) (SETQ X (MINUS X)))
  (GO LOP)
  OUT
  ... )

```

A loop is detected and the assignment analysis reports that the variables x and y may be assigned there. The variable y can get the value 3 and x can be assigned to an integer. Depending on this analysis, the variable information about x and y before the reduction of the loop will be that x is an integer and y can have either NIL or 3 as value.

After some reasoning, however, one can show that x can never be 1 (if we assume the function foo not to perform any side-effects on x and y) and therefore y never be 3. The correct reduction should be to eliminate the two and-expressions. This example shows that in the general case we need a deduction capability to prove properties about variables.

This is not included in this version of REDFUN. By this new scheme, however, a larger class of prog-expressions will be reduced in a satisfactory way.

5.1.6 Other features. Opening of functions has to be looked over and a more systematic implementation is desirable of the various ways a function can be opened. Beta-expansion and open specialization come from REDFUN (program B) and closed specialization from REDFUN'(version E). It would also be desirable to have an automatic procedure which can choose an appropriate way to open a function call depending on the arguments and the definition of the function.

The collapser part has to be extended, so it can handle cases described in section 4.1.1 in a correct way. What one really would like is a more general pattern matching system in order to describe the transformations one would like to perform in the code, but this will not be included in the REDFUN-2 generation of the system.

Extension of the number of function classes. A function can belong to several classes depending on the arguments to the function. The function car is treated as a pure if its argument is a list and as a reducer if the argument is a literal atom, in which case car performs an evaluation of a global variable.



## 5.2 PRINCIPAL CHANGES IN THE DESIGN OF THE NEW SYSTEM RELATIVE TO REDFUN

---

In this section we shall discuss the main changes in the design of REDFUN-2 relative to REDFUN. A number of concepts, which will be used in the rest of the report, will also be defined here.

The section 5.2.1 gives also some background of the design of the program structure used at the implementation of REDFUN.

5.2.1 Design of the program structure. The first version of REDFUN was a fairly straightforward and well-structured program. The code traversal and the various actions performed on the code were done through the main functions redform, redargs and redfun (described earlier in section 4.1.1). The closeness to the interpreter functions eval, evlis, apply etc. made these functions easy to understand. The functions in the substitution package followed the same pattern. So, when we process an expression we normally first recursively process its subexpressions, in principle the recursive decent method in compiling techniques (GRI71). Information is brought from a higher expression down to a subexpression through parameters and we cannot underneath from an expression reach its super-expressions.

The REDFUN-2 system follows in principle the same design, but compared to REDFUN information can be sent back from a sub-expression to higher expressions.

Another design criterium is that the traversal in the code is made as much as possible in a one-pass manner. This means that property extraction from the code is made parallel with the reduction. This cannot always be done, e.g. in a prog-ex-pression a sweep through the code has to be done in order to analyze the goto-structure and the assignments before the reduction is performed.

In the PRD-technique discussed in the REDFUN-report (BEC76) and in Sandewall (SAN76) a program is organized in a "data-driven" way where procedures associated with data items from the problem domain are stored on property lists. One advantage in structuring a program in this way is that it is easier to extend the program by adding new procedures. In REDFUN it was used for reducers and collapsers and in REDFUN-2 it is also used for the "semantic" routines, which are used to supply information about LISP functions.

5.2.2 Value-descriptors. In REDFUN, a variable from the processed code was stored in an association list (a-list) together with its value, if it was known, or together with the atom NOBIN. In REDFUN-2 the variable will be stored together with a value-descriptor. Such value-descriptors can be of 4 different types.

VALUES        to describe the values a variable can be bound to or be returned from a form. We will use the notation

$$X_{\text{values}} = \{A \ B \ C\}$$

meaning that x can have either A, B or C as value and no other values.

NOVALUES     to describe values a variable or a form cannot have and is written as

$$X_{\text{novalues}} = \{1 \ 2\}$$

with meaning that x cannot have 1 or 2 as value.

DATATYPE     to describe the datatype of those values, which can be bound to a variable or be returned from a form. We write this as

$$X_{\text{datatype}} = \text{integer}$$

NOBIN           to show that no information is available for  
a variable and is written as

$x_{\text{nobin}}$

A value-descriptor is simply implemented as a pair which first element describes the type and the second element the value(s). The values example above is represented as

(:VALUES . (A B C))

5.2.3 The q-tuple. In order to return information from a processed expression, from a called procedure in REDFUN-2 to its caller, a q-tuple will be returned. Such a tuple consists of six elements

< form, values, side-effect, truectxt, falsectxt, assigninfo >

where

form                   is the reduced expression

values                is a value-descriptor specifying the  
value(s) form can return when evaluated

side-effects        is a flag indication whether a side-effect  
can be performed or not in form when evaluated

truectxt             is an a-list of variables and value-  
descriptors where the value-descriptor  
represents information extracted for a  
variable from form which holds when form  
has been evaluated to true

falsectxt            as truectxt but which holds when form  
is evaluated to false

assigninfo          is an a-list of variables and value-  
descriptors for those variables which may  
be assigned in form. The value-de-  
scriptor describes the value(s) the form  
can be assigned to

When an ordinary form is processed by redform the arguments are first reduced through redargs. Redargs will now after every reduction of an argument check if there is any information in the q-tuple about assignments in the form which can affect the a-list before the next argument has to be reduced. When all arguments are reduced a new q-tuple is created for the form with information extracted from the arguments's q-tuples.

An example

Consider the expression

```
(COND ((EQ X Y) (F1 X))
      (T (F2 X)))
```

and a-list information that

```
Y_values = {A}
```

holds before the expression. When the eq-expression has been reduced following information is found in the q-tuple.

```
< (EQ X 'A),
  NIL,
  false,
  [x_values = {A}],
  [x_novalues = {A}],
  NIL >
```

Some remarks. The possible values returned from the eq-expression will not be stored in the q-tuple. The eq-expression stands in a predicate- position and the value is therefore of no interest here. When the f1-expression is reduced the a-list will contain

```
Y_values = {A} and x_values = {A}
```

and when the f2-expression is reduced

```
Y_values = {A} and x_novalues = {A}
```

No side-effects can occur in this form and that element in the q-tuple is NIL.

5.2.4 Semantic procedures. Pseudo-computations described in 5.1.2 and the property extraction of variable values described in 5.1.4 will be performed by semantic procedures which can be associated with every function. These procedures can be seen as a procedural description of some semantic property, which holds for the function. The system will contain a basic set of such procedures and the user should be able to supply additional procedures for system functions or functions he writes himself.

5.2.5 Redesign of the prog-reducer. The extended scheme to reduce prog-expressions described in 5.1.5, will cause a total redesign of the prog-reducer. The necessary programs for the analysis of the goto-structure and the assignments in the prog-expression and programs which clean up in the expression after the reduction (postprog-transformations) will be generated with the support by the PMG-system (RIS74). The traversal of the code in these programs follows the PMG-standard which makes it easier when there are user-defined functions which do not follow the standard argument evaluation.

5.2.6 Redesign of the substitution package. REDFUN contained a separate substitution package used when functions were beta-expanded in order to substitute formal arguments against actual ones. This caused some problems. When a new function with special argument structure is going to be processed by REDFUN code had to be inserted in the substitution package in order to perform the substitution correctly. There were also an efficiency problem because substitutions were made in parts of the code which later will be reduced away. In our new system these problems can be eliminated if we extend the a-list information also to include substitutions-descriptors, which is simply the form a variable has to be substituted for. The actual substitution will then be performed during the reduction process when the variable is encountered in the code. This causes that the substitution package can be thrown away in our new system.

### 5.2.7 Reduction in contexts. Consider the form

```
(CONS (PUT A I 'K) '(L M))
```

Usually, this form should be reduced to

```
(PROGN (PUT A I 'K) '(K L M))
```

However, if the form should appear in a context where its value is insignificant (for example a statement in a PROG expression, or in a non-terminal position in a PROGN expression), it can instead be simplified to

```
(PUT 'A 'I 'K)
```

It is also important to know the context a form appears in if extraction of variable properties should be done or not. If such properties never can be used they should not be calculated for efficiency reasons. Consider the expression

```
(COND ((EQ X 'A))
      (T (FOO X)))
```

If the cond-expression is a statement in a prog there is no use of extracting the information that x has the value A when the eq-expression is evaluated to true. It is either no use here of calculating the possible values the cond-expression can return.

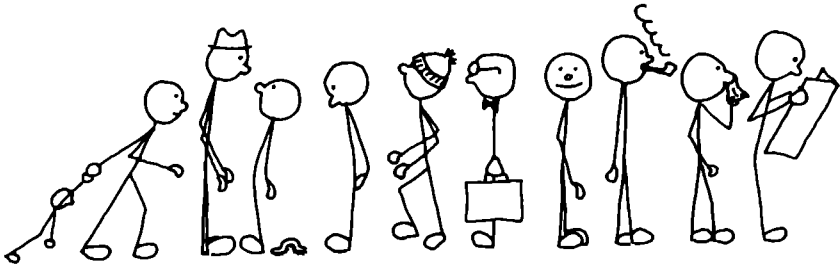
Therefore, for each form that REDFUN-2 encounters during its recursive descent into an expression, the program should know the context in which the expression is to be evaluated.

The following are the significant different cases:

novalue-context	no value(s) of the form has to be computed
value-context	compute value(s) if possible
boolean-context	it is sufficient to know if the form is true or false

There shall also be a way to distinguish if property extraction from the form should be performed or not described by the example above.

A similar context mechanism is found in a LISP compiler (URM77). There is a value- and a novalue-context. In a novalue context no code for returning a value will be generated. There are also contexts for branches in conditionals indication whether a jump will be performed or not if the predicate is evaluated to true or false.



"Q-TUPLE"





## 6. THE REDFUN-2 SYSTEM

### 6.1 OVERVIEW

In this chapter the new system REDFUN-2 is described. Every section covers a part of the system and describes its implementation. A number of examples are shown and discussed about restrictions and problems are also included.

In the description of the implementation we are trying not to go down to the LISP code level. When code is shown in this chapter it is done in order to give the reader an idea of the complexity of the code, rather than to actually describe it in detail. Appendix V contains the actual LISP code for some of the central functions, such as redform, redargs and tryapply, and for one of the reducers

A number of the examples used in this chapter are run through REDFUN-2, and the results are shown in appendix VI. These examples are marked in this chapter with (EX 4) corresponding to example 4 in the appendix.

In the examples in this chapter (and in the whole report) there are often calls to the procedures foo, fie, fum, f1, f2 etc. When nothing else is stated about them in an example we assume that these functions are "nice", meaning that they will not perform any side-effects or similar which may spoil the example.

Although the REDFUN-2 program is implemented in INTERLISP/20, the program processes code written in the INTERLISP 360/370 dialect. Only minor changes in the program are required to let it also process the INTERLISP/20 dialect. One difference between the dialects is the use of car to access a global value. In INTERLISP/20 this access has to be performed by the new function gettopval. Similarly, instead of using rplaca to assign a global value, one must use the new function settopval.

The following sections are included in this chapter

- 6.2 Extended range of information about variable values
- 6.3 Quoted-expressions
- 6.4 Computing values of forms
- 6.5 Extraction of variable properties
- 6.6 Handling of side-effects
- 6.7 Variable assignments
- 6.8 Extended function class authority
- 6.9 Reduction of prog-expressions
- 6.10 Opening of functions
- 6.11 Collapsers
- 6.12 Reduction in contexts

## 6.2 EXTENDED RANGE OF INFORMATION ABOUT VARIABLE VALUES.

6.2.1 A-list. In REDFUN there was only a simple a-list to hold the variables with known values. If the variable was bound to NOBIN, it had no known value and this fact was used to conceal other values for this variable bound on the a-list used in other environments.

In REDFUN-2 the a-list is extended to contain pairs of variables and value-descriptors. A value descriptor is represented either as NOBIN or as a pair

(type . val)

described by following table

type of value-descriptor	type	val
values*	:VALUE	a value
	:VALUES	a list of values
novalues	:NOVALUES	a list of values
datatype	:DATATYPE	a datatype

### Example

The a-list

```
((X . (:VALUES . (1 2 3))) (Y . NOBIN)
 (Z . (:DATATYPE . INTEGER))
 (Y . (:VALUE . T)) (V . (:NOVALUES . (A B C))))
```

carries the following information

X can have either 1, 2 or 3 as value  
 for Y nothing can be said  
 Z is of the datatype integer  
 V cannot have any of the values A, B or C

---

\* A distinction is made between the cases when there is a single value and several values.

6.2.2 Datatypes. The datatypes used in REDFUN-2 are the normal datatypes in INTERLISP, such as

SEXPR	S-expressions
ATOM	
LITATOM	literal atoms
INTEGER	
SMALLINTEGER	
LARGEINTEGER*	
FLOAT	floating point numbers
LIST	list structures
STRING	
ARRAY	

but do not include datatypes such as stack pointers, funarg blocks, function indicators etc. Other datatypes combined of those above are also included, such as

NUMBER for integers and floating point numbers

and variants of those, to which NIL can belong, such as

SEXPR-NOTNIL	S-expression except NIL
ATOM-NOTNIL	atom except NIL
LITATOM-NOTNIL	literal atom except NIL
LIST-NOTNIL	list structure except NIL

These datatypes are all true and are therefore useful in true/false - checks.

A variable known only as true resp. false is represented as  
(:NOVALUES . (NIL)) resp. (:VALUE . NIL)

---

\* In an early version of REDFUN-2 this type was called BIGINTEGER, but the INTERLISP 360/370 system was always obstinate to enter an infinite loop when it encountered the atom BIGINTEGER during reading.

Every datatype is also associated with a typefn procedure, which checks if its argument is of that type. We have for example

```
LIST : TYPEFN = *
  (LAMBDA (X)
    (OR (NULL X) (LISTP X)))

ARRAY : TYPEFN =
  ARRAYP

SEXPR-NOTNIL : TYPEFN =
  (LAMBDA (X)
    (AND X (OR (ATOM X) (LISTP X))))
```

Internally in the system there is also a negated version of these data types. For INTEGER there is an NEG-INTEGER, indicating that a variable with this datatype can not be an integer.

---

\* This notation means that the atom list under the property name typefn has the lambda-expression as property values.

6.2.3 Reductions of value-descriptors. Two value-descriptors can be combined either through an or-reduction or an and-reduction. This happens when more information has become available for a variable during the processing of the code.

Suppose

$$x_{\text{values}} = \{A\ B\ C\} \quad (*)$$

holds before the expression

```
(COND (L (SETQ X 'B))
      (LL (SETQ X 'D))
      (T NIL))
```

From this expression we shall extract new possible values for x and will receive

$$x_{\text{values}} = \{B\ D\}$$

The value-descriptor after the cond-expression will be calculated by the or-reduction

$$x_{\text{values}} = \{A\ B\ C\} \vee x_{\text{values}} = \{B\ D\}$$

resulting in

$$x_{\text{values}} = \{A\ B\ C\ D\}$$

If we instead have the expression

```
(COND ((MEMB X '(B C D)) (FOO X))
      (T (FIE X)))
```

and the same value-descriptor  $(*)$  holds for x before the cond-expression, the new value descriptor at the call to foo for x can be calculated by the and-reduction

$$x_{\text{values}} = \{A\ B\ C\} \wedge x_{\text{values}} = \{B\ C\ D\}$$

resulting in

$$x_{\text{values}} = \{B\ C\}$$

and at the call to file it is calculated by

```
x_values = {A B C} ^ x_novalues = {B C D}
resulting in
x_values = {A}
```

In some reductions, however, information will be lost when datatypes are involved. The following expression

```
x_datatype = integer v x_values = {3 A NIL}
will be reduced to
```

```
x_datatype = atom
```

Another example is that there is no way in this scheme to express that a variable can be bound only to values, which are either floating-point numbers or literals. There is no such combined datatype in the system. A reduction of

```
x_datatype = float v x_datatype = litatom
will result in
```

```
x_datatype = atom
```

The information that the variable cannot be bound to an integer is then lost.

There is a table containing all datatypes showing the relationships between them and which is used when reduction are performed. The table contains also all subdatatypes for a datatype. The datatype NUMBER has the subdatatypes INTEGER, SMALLINTEGER, LARGEINTEGER and FLOAT.

A reduction between datatypes is called a d-reduction.

This scheme for handling datatypes is appropriate in most cases, although one often, in a non-declarative language as LISP uses, for example, a temporary variable for different purposes in the code and therefore lets it hold objects of different datatypes.

The reduction schemata of the various types of value-descriptors is finally shown in two tables.

	$\beta$ values	novalues	datatype
$\alpha$ values	values $\alpha \cup \beta$	-	-
novalues	novalues $\alpha - \beta **$	novalues $\alpha \cap \beta$	-
datatype	datatype d-reduce $\alpha$ and the datatypes of the elements in $\beta$ .	novalues those ele- ments in $\beta$ which are not of the datatype $\alpha$ . If no such element exists a nobin value- descriptor is returnd.	datatype d-reduce $\alpha$ and $\beta$ .

Table 1 or-reduction \*

	$\beta$ values	novalues	datatype
$\alpha$ values	values $\alpha \cap \beta$	-	-
novalues	values $\beta - \alpha$	novalues $\alpha \cup \beta$	-
datatype	values those ele- ments in $\beta$ which are of datatype $\alpha$ .	datatype $\alpha$	datatype d-reduce (subdata- types of $\alpha$ ) $\cap$ (sub- datatypes of $\beta$

Table 2 and-reduction \*

\*, \*\* see next page



These tables should be read as follows. Suppose we want to reduce

$$x_{\text{novalues}} = \alpha \vee x_{\text{values}} = \beta$$

In the or-reduction table under values/novalues we find this to be reduced to

$$x_{\text{novalues}} = \alpha - \beta$$

The expression

$$x_{\text{datatype}} = \alpha \wedge x_{\text{values}} = \beta$$

is reduced through the and-reduction table under values/datatypes to

$$x_{\text{values}} = \text{"those elements in } \beta \text{ which are of datatype } \alpha \text{"}$$

and with  $\alpha = \text{literalatom}$  and  $\beta = \{A B 1 1.5\}$  we shall get

$$x_{\text{values}} = \{A B\}$$

Remark. And-reduction of a novalues and a datatype, ie

$$x_{\text{novalues}} = \alpha \wedge x_{\text{datatype}} = \beta$$

is choosen to be reduced to

$$x_{\text{datatype}} = \beta$$

It could also be reduced to

$$x_{\text{novalues}} = \alpha$$

In both cases we loose information about the variable and it is not obvious which of them is the most appropriate. If the later reduction would be desirable in some application it is trivial to change the program.

\* The or-reduction and the and-reduction are commutative operations and therefore only the lower half of the tables are filled in.

\*\*  $\alpha - \beta$  is the set difference between  $\alpha$  and  $\beta$

### 6.3 QUOTED-EXPRESSIONS

The q-tuple, which holds a reduced form and information extracted from it, is simply represented as a list, with the atom QUOTED as the first argument. Such a list is called a quoted-expression. The structure is

```
(QUOTED form values side-effect truectxt falsectxt assigninfo)
```

where form is the reduced form,

values a value descriptor,

side-effect the atom :SIDE if side-effects occur, otherwise NIL,

truectxt an a-list of variables and value descriptors, for information which holds when the form is evaluated to true

falsectxt as above, but which holds when the form is evaluated to false

assigninfo an a-list of variables and value descriptors for those variables which may be assigned in the form.

A quoted-expression is only created if there is any information available to store there.

## Example

Suppose we want to reduce following form

```
[AND (EQ X Y)
      (COND
        ((NUMBERP X)
         (FOO X))
        (T (SETQ Z (FIE X))
            (SETQ V (QUOTE B))
            (EX 1))
      )]
```

with the a-list

```
((Y . (:VALUE . A)))
```

We assume also foo and fie not to perform any side-effects.

Following quoted-expression will be the result

```
(QUOTED [AND (EQ X (QUOTE A))
              (PROGN (SETQ Z (FIE (QUOTE A)))
                     (SETQ V (QUOTE B))
                     (:VALUES NIL B)
                     :SIDE
                     ((V :SETQVALUE :VALUE . B)
                      (Z . NOBIN)
                      (X :VALUE . A))
                     ((X :NOVALUES A))
                     ((V :ADDVALUE :VALUE . B)
                      (Z :ADDVALUE . NOBIN)))
            ]])
```

Explanations for each element in this expression

- Some reductions could be performed. The cond-expression is reduced to its false branch.
- Possible values from this reduced form is NIL and B.
- The assignments caused the side-effect flag to be set.

\* When output is printed directly by the LISP system the pairs are not so clearly written. Remember that the expressions

```
(:VALUES NIL B)
```

and

```
(:VALUES . (NIL B))
```

are identical.

- If the form is evaluated to true we can derive that x has the value A, z has an unknown value and y has the value B. The :SETQVALUE in y's value-descriptor indicates that y has been assigned in the form to that value.
- If the form is evaluated to false we can derive that x cannot have the value A.
- In the form, y and z may be assigned, y to the value B and z to an unknown value. The :ADDVALUE in these value-descriptors indicates that the assignments may be performed. If :ADDVALUE was not there the assignments will always be performed when the form is evaluated.

## 6.4 COMPUTING VALUES OF FORMS.

6.4.1 Application of function. For pure-functions the operation application of function has been extended. If all arguments to a pure-function are known and no side-effects disturb them, then the function will be applied to its arguments. This operation can also be done when the arguments are of the values-type, when we know all the values an argument can be bound to. If we have

```
(ADD1 X)
```

and x's value-descriptor is

```
(:VALUES . (3 4 5))
```

an evaluation of add1 will be done on each possible argument value and the results will be 4,5 and 6.

This can also be performed for a function of two or more arguments. We must then apply the function to all the combinations of argument values which can appear. Suppose foo is defined as

```
(LAMBDA (X Y Z) (PLUS X (TIMES Y Z)))
```

and we perform

```
redform[(LESSP (FOO A B C) 6),((A . (:VALUES . (3 4 5)))
                                (B . (:VALUE . 2))
                                (C . (:VALUES . (2 -2))))]
```

we shall have 6 (3 x 1 x 2) argument combination to compute

```
(FOO A B C)
```

which will result in the values -1,0,1,7,8 or 9, and

```
(LESSP (FOO A B C) 6)
```

is consequently calculated for 6 (6 x 1) argument combinations with the result NIL or T.

A problem arises here. The combinations of arguments can be very many or the evaluation of the function very expensive, so we cannot afford to compute them. This is solved in REDFUN-2 since the user can for every pure-function give the maximum number of combinations it may have, in order to allow the computation to be done. A default maximum for functions can be set and is actually in the present system set to 20. Another way could be to let the system ask the user what to do in this situation, for situations can of course arise where these computations can be the crucial ones in order to be able to perform reductions in the program. The best thing would of course be if we had a smart analysing program, which could tell when and where it is worth spending time on such computations.

In some cases this evaluation with all argument combinations can be made more efficient by a semantic procedure. Suppose

```
(EQ X Y)
```

and that x and y can have 50 different values each. In order to find out if this eq-expression will always be evaluated to NIL, instead of applying eq on 2500 argument pairs, we can make a check if an intersection of x's and y's values will be empty or not. If it is empty no x and y are equal and we can therefore deduce the value always to be NIL.

For some functions whose arguments are of the novalues-type similar computation can be done, such as for

```
(ADD1 X)
```

and when x's value descriptor is

```
(:NOVALUES . (2 3 4))
```

we can then compute the value to

```
(:NOVALUES . (3 4 5))
```

This can only be done for functions where there is an one-to-one mapping between the arguments and the value. Actually there are not so many for which this holds. In the present system we have only included add1, sub1, and minusp and they are called evalnovaluefns.

The functions cons and list also belong to this category but there is probably not much point in performing this computation for them. The user can of course extend this category with other functions.

#### 6.4.2 Computing values from conditionals and logical functions.

For conditionals it is possible to deduce what value(s) can be returned, if we have enough knowledge about the value(s) for the different branches.

For a cond-expression its value-descriptor can be calculated in the following way. Suppose we have

```
(COND (p1 e1) (p2 e2) ... (pk) ... (pn en))
```

and that valdescr(e<sub>j</sub>) is the value-descriptor for e<sub>j</sub>.

- a. Start with the empty set v.
- b. For each p<sub>i</sub>, which is not known as false, add if e<sub>i</sub> exists valdescr(e<sub>i</sub>) otherwise valdescr(p<sub>i</sub>) to v. If p<sub>n</sub> is not known as true add the descriptor (:VALUE . NIL) to v. Make an or-reduction of v.
- c. The remaining element is the value-descriptor for the cond-expression.

We can have cases where side-effects in predicates prevent us from removing false predicates, such as in

```
(COND ((FOO X) 10)
      ((SETQ Z NIL))
      (T 12))
```

Here the possible values are 10 and 12.

For an or-expression and selectq-expression it is simple to perform an or-reduction of the different argument's resp. cases's value-descriptors.

For an and-expression

```
(AND e1 e2 ... en)
```

an or-reduction is done on `valdescr(en)` and `(:VALUE . NIL)`

Consider

```
(AND (FOO X) (FIE X) 'A)
```

and we know nothing about foo and fie, we shall receive the values A or NIL.

6.4.3 Pseudo-computations. In many cases we can perform pseudo-computations of functions when we only have partial knowledge about its arguments or when application of function cannot be performed because side-effects are involved. By using a pseudo-computation a value can be calculated for a form although we have not actually evaluated it.

For some functions we know the value by its definition. The function put returns its third arguments as value, so if we know the value(s) that argument can have, we also know the value(s) which will be returned, for example

```
(PUT A P 'VAL)
```

will result in

```
(QUOTED (PUT A P 'VAL) (:VALUE . VAL) :SIDE)
```

The expression is also marked to perform a side-effect when it is evaluated.

A number of sematic procedures, which perform various type of pseudo-computation can be associated with a function.

The function put will have a valuefn-procedure,

```
([LAMBDA (C F V)
  (GETVALUES V))
```

where getvalues fetches the values-element from a quoted-expression.



A function foo, which returns the constant 'A can be associated with

```
FOO : VALUEFN =
  [LAMBDA (X)
    (MAKE:VALUE (QUOTE A)]
```

where make:value creates a value-descriptor.

For pure-functions we can anticipate that evaluations of a function for all argument combinations will be performed by using a valuefn-procedure instead (also discussed in 6.4.1). This is useful in cases, such as when we have

```
(EQ X Y)
```

and x and y can each have several different values. It is then interesting to know if this expression will always be evaluated to false and instead of actually performing all evaluations of the form we can calculate it by a procedure defined as

```
EQ : VALUEFN = *
  [LAMBDA (X Y)
    (COND ((INTERSECTION (:VALUES X)
                          (:VALUES Y))
           NIL)
          (T (MAKEFALSEVAL)]
```

If the intersection of x's and y's values is empty, then we can deduce that the expression always will be false. The function makefalseval, makes the value-descriptor representing the false value.

---

\* If NIL is returned from a semantic procedure it was not applicable, otherwise a value-descriptor is returned.

## Examples

```
redform[(EQ X Y),((X . (:VALUES . (A B))) (EX 2)
              (Y . (:VALUES . (C D)))))]
```

```
= NIL
```

```
redform[(EQ X Y),((X . (:VALUES . (A B))) (EX 3)
              (Y . (:VALUES . (A C)))))]
```

```
= (EQ X Y)
```

Consider the same example again

```
(EQ X Y) (EX 4)
```

and an a-list, which looks like

```
((X . (:VALUES . (B C))) (Y . (:NOVALUES . (A B C))))
```

We can also here deduce that the eq-expression will always be false. The function eq will also have a novaluefn-procedure, which performs that computation. The procedure is defined as

```
[LAMBDA
  (A B)
  (PROG
    ((A (VALUEFY A))
     (B (VALUEFY B))
     (VAL-NOVAL (VAL-NOVAL? A B)))
    (RETURN
     (SELECTQ
      VAL-NOVAL
      (VAL-NOVALS (AND (MEMB A B)
                       (MAKEFALSEVAL)))
      (VALS-NOVALS (AND (EVERY A
                             (F/L (X)
                                   (MEMB X B)))
                       (MAKEFALSEVAL)))
      (NOVALS-VAL (AND (MEMB B A)
                      (MAKEFALSEVAL)))
      (NOVALS-VALS (AND (EVERY B
                             (F/L (X)
                                   (MEMB X A)))
                      (MAKEFALSEVAL)))
      NIL])
```

where valuefy and val-noval? decide which combination of :value, :values and :novalues the value-descriptors of the arguments belong to. This procedure is invoked when at least one argument is of the novalues-type.

There is also a novaluefn-procedure for memb. If we arrive at a situation where

```
(MEMB X '(A D))
```

```
(EX 5)
```

and x's value-descriptor is

```
(:NOVALUES . (A B C D))
```

we can deduce that the memb-expression will be false.

The INTERLISP predicates, which test datatype authority, such as listp, arrayp, fixp etc. have an associated datatypefn-procedure which simply checks if the argument to such a predicate will result in a true\* or false value. If we have

```
(FIXP X)
```

and x is a large integer, the procedure returns a true value-descriptor. If it is a literal atom a false value-descriptor is returned or finally if x is a number nothing can be said about the value.

---

\* Actually in INTERLISP, some predicates return T and other return its argument as the true value.

Finally there is a semantic procedure, the tryapplyfn-procedure, to cover all other possibilities. This is used in the same way as the collapsers and the user provides contributions which are inserted in the procedure by an advise-technique. The procedure is associated with a pure-function, and the arguments are passed over to the procedure, where any computation can be performed, depending on the arguments. It is used for example on eq to check if its arguments are of different and disjoint datatypes, in which case we know that eq will be false.

EQ : TRYAPPLYFN =

```

(LAMBDA (CTXT X Y)
  (PROG (DX DY)
    (RETURN (COND ((AND (SETQ DX
                          (GET:DATATYPE
                           X))
                        (SETQ DY
                          (GET:DATATYPE
                           Y))
                     (DISJOINTDATATYPE
                      DX DY))
                  (MAKEFALSEVAL))
          (T NIL)

```

The expression

(EQ X 'A)

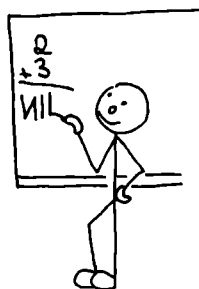
(EX 6)

with

((X . (:DATATYPE . INTEGER)))

will result in

NIL



"PSEUDO COMPUTATION"

6.4.4 The tryapply-procedure. The application of function to constant arguments and the invocation of these semantic procedures are performed in REDFUN-2 by the function tryapply. Here we shall summarise the different actions which this function performs for a pure LISP function.

Tryapply has three arguments, a function fn, an argument list args and a context\* to reduce in rctxt. The arguments have been reduced through redargs. The following cases can now appear:

- a. An empty argument list. The function will be evaluated by

```
res := apply[fn,NIL]
```

and the value from

```
qquote[res]
```

is returned. The function qquote puts res in a quote-expression, except for NIL, T and numbers.

The arguments will be classified into one of the following classes:

- b. "allqwoted". All arguments have a single known value and no side-effects will interfere, so an evaluation can be done and the value from

```
qquote[eval[cons[fn,args]]]
```

will be returned.

---

\* That context is described in section 6.12

- c. "allsinglevalue". As b. but side-effects are involved among the arguments. A progn-expression of the arguments, in which side-effects will be performed at evaluation, is created and the value of the expression will be calculated, when arguments with side-effects have been replaced by their values. This value will then be concatenated at the end of the progn-expression. An example of this is the form

```
(LIST (SETQ X Z) V (PUT A P X))
```

in which list is supposed to be pure and with the a-list

```
((Z . (:VALUE . ADAM))) (V . (:VALUE . DAVID)))
```

The result will be returned as

```
(QUOTED (PROGN (SETQ X (QUOTE ADAM))
               (PUT A P (QUOTE ADAM))
               (QUOTE (ADAM DAVID ADAM))))
(:VALUE . (ADAM DAVID ADAM))
:SIDE
... )
```

- d. "allknownvalues". The arguments's value-descriptors are of values-type, i.e. we know all the values the arguments can have and side-effects can be involved. The following cases can appear here:

- d1. The number of argument combinations exceeds the maximum combinations allowed for the computations to be done (see 6.4.1) in which case just the value of the form

```
cons[fn,args]
```

is returned.

- d2. Application of the function to every argument combination will be performed. We can then get one single or several values from these computations. If we get one value and there are no side-effects in the arguments, an expression will be returned as in b. and with side-effects a progn-expression as in c.

## Examples

```
(LESSP X Y)
```

with a-list

```
((X . (:VALUES . (1 2))) (Y : (:VALUES . (5 7 8))))
```

will result in

```
T
```

but

```
(LESSP X (PUT A P Y))
```

with the same a-list will result in

```
(QUOTED (PROGN (PUT A P Y) T) (VALUES . T) :SIDE))
```

d3. Finally if there are several results

```
(QUOTED cons[fn,args] (:VALUES . (all values)) ...)
```

- e. "allnovalues". If all arguments involved are of novalues-type and the function is defined to be an evalnovaluefn, it will be applied to all argument combinations and the results will be the values which the form cannot have. This was described earlier in 6.4.1. An example

```
(ADD1 X)
```

with the a-list

```
((X . (:NOVALUES . (1 2 3))))
```

will result in

```
(QUOTED (ADD1 X) (:NOVALUES . (2 3 4)))
```

- f. If none of the above computations has been done successfully then sematic procedures are invoked.

f1. If at least one argument is of novalues-type the novaluefn-procedure is called.

f2. If at least one argument is of the datatype-type the datatypefn-procedure is called.

f3. If none of these was successful then the tryapplyfn-procedure is called.

g. If the function fn returns values of a certain datatype, it is notified in the quoted-expression, such as for

```
(PACK L)
```

which will result in

```
(QUOTED (PACK L) (:DATATYPE . ATOM))
```

The datatype of the result can for some functions depend on the datatypes of their arguments, i.e. plus, difference etc. If the arguments are all integers the result will also be an integer, but if one of the arguments is a floating point number the result will be so too. Different cases can therefore occur

```
(PLUS 2 X)
```

in which x is an integer, will result in

```
(QUOTED (PLUS 2 X) (:DATATYPE . INTEGER))
```

and

```
(PLUS 2.5 X 3)
```

in which nothing is known about x will be

```
(QUOTED (PLUS 2.5 X 3) (:DATATYPE . FLOAT))
```

and finally

```
(PLUS 2 X)
```

in which we know nothing about x will be

```
(QUOTED (PLUS 2 X) (:DATATYPE . NUMBER))
```

A specialization of the function plus can also be done to give either iplus or fplus, as in the examples above, by collapse rules.



- h. If all different values a function can return are known and stored as `fn:values`, a value-descriptor is created by these values.

Example

```
(GRATERP X Y)
```

results in

```
(QUOTED (GREATERP X Y) (:VALUES . (T NIL)))
```

- i. In all other cases

```
cons[fn,args]
```

is returned.

#### 6.4.5 Summary of the semantic procedures.

valuefn invoked by expr-functions\* and sideexpr-functions\*, e.g. put and print. Used when a value can be calculated from such a function. Also used by pure-functions to anticipate evaluations of functions applied to several argument combinations when that calculation can be performed in other ways.

novaluefn invoked by pure-functions, when at least one argument is of novalues-type. Useful for eq and memb for example.

datatypepfn invoked by a pure function, when at least one argument is of datatype-type. The INTERLISP predicates which test datatype authority have such a procedure.

tryapplyfn invoked by pure-functions.

These categories and classes of semantic procedures must not be seen as an exhaustive classifying of LISP functions. They only demonstrate some properties of these functions and some cases which it is useful to separate in this application.

---

\* expr and sideexpr are function classes described in section 6.8.1.

## 6.5 EXTRACTION OF VARIABLES PROPERTIES

6.5.1 True- and false-branches. In conditionals, such as cond and selectq and in logical functions, such as and and or, it is possible to extract properties of variables occurring in the predicates and forms respectively which hold if we assume that the predicate or form will be true or false.

If we have

```
(AND (EQ X 'A) (FOO X) ...)
```

we know that if the first form is true then x in the second form will have the value A. We call the remaining forms after the first one to be in a true-branch with respect to the first form. There can also be a false-branch.

In a cond-expression these branches are defined as follows.

```
(COND (p1 e1)
      (p2 e2)
      .
      .
      (pn en))
```

The true-branch with respect to p<sub>i</sub> is the expression e<sub>i</sub>, i.e. the form(s) which will be evaluated if p<sub>i</sub> is true. The false-branch with respect to p<sub>i</sub> consists of the succeeding clauses.

```
(pi+1 ei+1)
.
.
.
(pn en)
```

In a selectq-expression

```
(SELECTQ case
  (p1 e1)
  (p2 e2)
  .
  .
  .
  en)
```

the true-branch with respect to case=p<sub>i</sub> (or case ∈ p<sub>i</sub>) is the expression e<sub>i</sub> and the false-branch the expressions e<sub>j</sub>, j=i+1,n.

In an and-expression

```
(AND e1 e2 ... en)
```

the true-branch with respect to e<sub>i</sub> is the rest of the expressions e<sub>j</sub>, j=i+1,n. The false-branch is empty.

In an or-expression

```
(OR e1 e2 ... en)
```

the false-branch with respect to e<sub>i</sub> is the rest of the expressions e<sub>j</sub>, j=i+1,n. The true-branch is empty here.

6.5.2 The truectxt- and falsectxt-elements. The information which has been extracted from a form, i.e. pair of variables and value-descriptors, which holds in the true-branch with respect to the form, is stored in the truectxt-element in the quoted-expression. Information which holds in the false-branch is stored in the falsectxt-element.

The or-expression in

```
(COND ((OR (EQ X 1) (EQ X 2)) (F1 X))
      (T (F2 X)))
```

will after the reduction appear as

```
(QUOTED (OR (EQ X 1) (EQ X 2))
        NIL
        NIL
        ((X . (:VALUES . (1 2))))
        ((X . (:NOVALUES . (1 2))))
        NIL)
```

In the true-branch the truectxt-element gives that x has the value 1 or 2, and in the false-branch the falsectxt-element gives that x cannot be 1 or 2.

6.5.3 Example. Let us follow an example<sup>\*</sup>

```
(COND ((EQ X 'A) (F1 X))
      ((MEMB X '(A B)) (F2 X))
      ((EQ X 'D) (F3 X))
      (T (F4 X)))
```

(EX 7)

Suppose we also know that

$X_{\text{values}} = \{A\ B\ C\}$

holds before the cond-expression.

---

<sup>\*</sup> We assume f1, f2 etc. not to perform any side-effect on the variable x.

From the first eq-expression we can deduce that in its true-branch x must have the value A. In its false-branch a specialization is performed of what is known about x before the cond-expression and the fact that x cannot have the value A, which results in the and-reduction

$$X_{\text{values}} = \{A\ B\ C\} \wedge X_{\text{novalues}} = \{A\} \rightarrow X_{\text{values}} = \{B\ C\}$$

The memb-predicate will transfer to its true-branch the information that x must have the value B, which is also done after an and-reduction

$$X_{\text{values}} = \{B\ C\} \wedge X_{\text{values}} = \{A\ B\} \rightarrow X_{\text{values}} = \{B\}$$

To the false-branch is transferred the information that x will have the value C, so the next eq-expression will always be false and can be eliminated. The whole cond-expression can consequently be reduced to

```
(COND ((EQ X 'A) (F1 'A))
      ((MEMB X '(A B)) * (F2 'B))
      (T (F4 'C)))
```

---

\* A collapsing of the memb-expression to

```
(EQ X 'B)
```

might be preferable. Such collapsing can only be done if it is not important what true-value will be returned. The memb-expression will return (B) but the eq-expression T.





The function make:datatype creates a value-descriptor of the datatype-type. For

```
(LITATOM X)
```

the following ctxt-descriptor will be obtained

```
(:CTXT ((X . LITATOM)) (X . NEG-LITATOM))
```

6.5.5 Logical functions and conditionals. For logical functions and conditionals the respective reducer is responsible for collecting the information about properties of variables which holds in the true- and false-branch, i.e. the truectxt-element and the falsectxt-element in the quoted-expression.

6.5.5.1 And- and or-expressions. An and-expression is true if every argument is also true and the truectxt-element can then be derived from the truectxt-element for each argument.

In the expression

```
(AND (EQ X 'A) (MEMB Y '(1 2 3)))
```

we can derive that x is A and y one of 1, 2 or 3 when the whole expression is true. What can be derived when it is false? In this case we cannot know which of the two arguments was calculated to the value NIL and cannot derive anything about either x or y to be used in the false-branch. The same happens with or-expressions. For them we can derive information when it is false, all arguments have then been evaluated, but cannot always extract information to be used in its true-branch for the same reason that we cannot know which of the arguments was calculated to true. But in some cases, however, information can be derived. In the expression

```
(OR (EQ X 'A) (EQ X 'B))  (*)
```



we will of course know that x will be either A or B if the whole expression is evaluated to true. If a truectxt-element is available for a variable in every argument (which is not known always to be false) in the or-expression a truectxt-element can be calculated for the whole or-expression and is calculated by an or-reduction for that variable's value-descriptors which holds in the true-branch for each argument. The same holds for the and-expression when it is calculated to false.

6.5.5.2 Cond-expression. For cond-expressions it is a little more complex to derive this information, both to find those variables for which something can be said, how the information is extracted from the cond-expression and to calculate the value-descriptor which holds for the variable when the expression is calculated to true and false respectively.

Let us follow an example.

```
(COND ((EQ X 5) T)
      ((EQ X 7) NIL)
      (T Y))
```

(EX 8)

We assume no information about x and y is available at the entry of the expression. For the variable y nothing can really be said. The cond-expression can be both true and false in the first two clauses regardless of the value of y. For the variable x however, information can be derived to be used both in its true- and false-branch. The cond-expression can be true either in the first clause or in the third one. From these two clauses the following value-descriptors for x hold

$X_{\text{values}} = \{5\}$

from the first clause and

$X_{\text{novalues}} = \{5\ 7\}$

from the last one.

The value descriptor which holds generally for x in the true-branch can then be deduced from these two by an or-reduction which will result in

$$x_{\text{novalues}} = \{7\}$$

i.e. the conclusion is that x cannot have the value 7 if the whole cond-expression is evaluated to true.

By the same reasoning for x when the whole cond-expression is false, the or-reduction

$$x_{\text{values}} = \{7\} \vee x_{\text{novalues}} = \{5\ 7\} \rightarrow x_{\text{novalues}} = \{5\}$$

gives the result that x cannot have the value 5 in its false-branch.

6.5.5.3 Algorithms. Two algorithms are used to extract variables properties in a cond-expression and here we shall show those for extraction information for variables while holds in the true-branch. The corresponding algorithms for the false-branch are very similar.

#### Algorithm I

To find those variables, for which a truectxt-element can be created.

#### Algorithm II

To collect for such variables value-descriptors in the cond-expression from which the resulting value-descriptor to be inserted in the truectxt-element can be calculated.

The reason for having two separate algorithms is that it is more economical first to scan through the clauses in the cond-expression and find those variables for which it is worth collecting value-descriptors and performing reductions. The latter algorithm is much more resource consuming than the first one.

We assume

(COND  $c_1$   $c_2$  ...  $c_n$ )

where  $c_i$  is a clause either  $(p_i \ e_i)$  or  $(p_i)$ .

A true clause is a clause, which will always return a true value. A false clause will always return a false value. The two procedures trueclause and falseclause check this.

The procedure pred returns the predicate in a clause and the procedure expr returns either  $e_i$  if the clause is of type  $(p_i \ e_i)$  or  $p_i$  if of type  $(p_i)$ .

The procedures truectxtvars and falsectxtvars return those variables for which information is available to be used in the true- resp. false-branch.

#### Algorithm I

- a. Initialize the variables truebranch and falsebranch to false. They are used to flag if a true resp. a false clause has been encountered.  
Set vars to the empty set.
- b. for  $i := n$  to  $1$  do  
comment the clauses are processed backwards;  
 $p_i := \text{pred}(c_i)$ ;  
 $e_i := \text{expr}(c_i)$ ;  
if falseclause( $c_i$ )  
   then if falsebranch = false  
     then falsebranch := true;  
        $\text{vars} := \text{vars} \cup \text{falsectxtvars}(p_i)$   
     else  $\text{vars} := \text{vars} \cup \text{truectxtvars}(p_i)$   
   else if truebranch = false  
     then truebranch := true;  
        $\text{vars} := \text{truectxtvars}(e_i) \cup \text{truectxtvars}(p_i)$   
     else  $\text{vars} := \text{vars} \cap (\text{truectxtvars}(e_i) \cup$   
        $\text{truectxtvars}(p_i))$ ;  
     if not(trueclause( $c_i$ )) then falsebranch := true;

- c. The variables, for which a truectxt-element can be created are to be found in vars.

### Algorithm II

Var is the variable from vars after step c. in algorithm I. For each clause c<sub>i</sub>, which is not a false clause, calculate the value-descriptor fro var, which holds after expr(c<sub>i</sub>). Collect those value descriptors and perform an or-reduction and the result is the wanted value-descriptor.

Let us follow the algorithms for the example

(COND ((EQ X 3) NIL)	clause 1	
((EQ X 5) (EQ Y 5))	2	(EX 9)
((EQ Y 3) T)	3	
(T NIL))	4	

Algorithm I will process the clauses backwards and we shall repeat step b. 4 times. The values of the variables in the algorithm are shown in the following table after that a clause has been processed.

	truebranch	falsebranch	vars
clause 4	false	true	NIL
3	true	true	(Y)
2	true	true	(Y)
1	true	true	(Y X)

The algorithm II will consequently be performed for both variables x and y. A true value from the cond-expression can be received from clauses 2 and 3 and the next table shows the value-descriptors for x and y in these clauses

clause 2	X <sub>values</sub> = {5}	Y <sub>values</sub> = {5}
3	X <sub>novalues</sub> = {3 5}	Y <sub>values</sub> = {3}
or-reduc- tion	X <sub>novalues</sub> = {3}	Y <sub>values</sub> = {3 5}

When the cond-expression is true it holds that x cannot have the value 3 and that y will have one of the values 3 or 5.

6.5.6 Own "datatypes". With this scheme the user can himself introduce his own "datatypes" for variables and be able to remove redundant datatype checks for example. Suppose we have in an application a new datatype which we call persrecord, implemented as a free property list headed by a datatype marker PERSREC. We will then have a predicate persrecp, which tests if its argument is of that datatype. It can simply be defined as

```
(LAMBDA (L) (AND (LISTP L) (EQ (CAR L) 'PERSREC)))
```

A normal technique is then to introduce a number of small primitive functions to access or test fields in such records. To be careful one should always check in these procedures that the datatype they operate on is correct and otherwise make an error-message or perform some other action. We can for example have a function getage to access the age-property of such record, and this can be defined as

```
(LAMBDA (L) (COND ((PERSRECP L) (GET (CDR L) 'AGE))
                  (T (ERROR-ROUTINE-FOR-WRONG-DATATYPE))))
```

In a production system it can be expensive with all these datatype checks and opening these primitive procedures would probably be desirable so redundant checks and superfluous code for error-routine calls could be removed.

In REDFUN-2 we can declare the persrecp function as pure and associate a datatypepfn- and a ctxtfn-procedure

We can define the datatypefn-procedure as

```
(LAMBDA (L)
  (COND ((EQ (GET:DATATYPE L) 'PERSREC) (MAKETRUEVAL))
        ((EQ (GET:DATATYPE L) 'NEG-PERSREC) (MAKEFALSEVAL))
        (T NIL)))
```

and the ctxtfn-procedure

```
(LAMBDA (RCTXT L)
  (COND ((VARIABLE L) (MAKE:CTXT (LIST
                                   (CONS (VARIABLE L)
                                           (MAKE:DATATYPE 'PERSREC)))
                                   (LIST
                                   (CONS (VARIABLE L)
                                           (MAKE:DATATYPE 'NEG-PERSREC))))
        (T NIL)))
  RCTXT)
```

Consider now cases where the code is

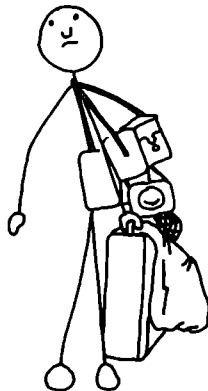
```
(AND (PERSRECP L) (GETAGE L))
```

which after opening of getage will result in

```
(AND (PERSRECP L) (GET (CAR L) 'AGE))
```

In getage we know that l has already been checked to be of the right datatype so the desirable reduction could be performed.

"SIDE EFFECTS"



## 6.6 HANDLING OF SIDE-EFFECTS.

Special care must be taken when side-effects occur in the code. A form which can perform a side-effect when it is evaluated will be marked in the quoted-expression. Example

```
(QUOTED (PUT A B C) NIL :SIDE)
```

We can distinguish some different cases.

- Side-effects among arguments to a pure-function. This has been discussed in 6.4.4. The principle was that if the values of the arguments are known, the value of the whole expression can be calculated and a progn-expression will be created by those arguments with side-effects together with the value.

Suppose

```
(MEMB (SETQ X 'A) (CONS (SETQ Y 'C) Z)) (EX 10)
```

and z's value-descriptor

```
(:VALUE . (B A))
```

The cons-expression can be calculated and reduced to

```
(PROGN (SETQ Y 'C) '(C B A))
```

The whole expression will then be

```
(PROGN (SETQ X 'A) (SETQ Y 'C) '(A))
```

- Functions performing side-effects. These are normally declared as a sideexpr or a sidefexpr. For a sideexpr function the arguments will first be reduced and by the valuefn-procedure a pseudo-computation can be performed to obtain its value if enough information about the arguments is known. A sidefexpr function will not be affected and is only marked to perform side-effects. For special cases one can have a reducer-property and perform the necessary processing oneself.
- Side-effects depending on variable assignments are described in next section 6.7.

## 6.7 VARIABLE ASSIGNMENTS.

Assignments must be taken care of in a systematic way. A variable which has had known value(s) can be assigned to a new value which can be known or unknown, or a new variable can be assigned known value(s). The a-list in REDFUN which holds the value-descriptors of such variables must be updated.

When assignments are done in conditionals, logical functions and prog's the problem of keeping track of these assignments is rather complex. Here follow some cases which can be distinguished. Assignments in prog-expressions are described in 6.9.

6.7.1 Assignments in arguments to a function of pure, ~~expr~~ or sideexpr-type. If an assignment will always be performed when the arguments are evaluated, the assignment must affect the a-list in REDFUN when processing the remaining arguments in that form and the succeeding forms after that function call.

Example

```
(CONS (SETQ Z 5) (LIST (SETQ X Y) (FOO X Z)))
```

If we do not know anything about y and foo performs no assignments on z and x, we can deduce that when the foo-expression is processed the a-list contains

```
((X . NOBIN) (Z . (:VALUE . 5)) ....)
```

The same a-list will be used for the forms following this cons-expression.

6.7.2 Assignments inside conditionals and logical functions.

In such expression we can not be sure that the assignment will be performed, so we must keep track of the different cases which can occur. If an assignment is done in a branch in a cond-expression that assignment is only performed if its corresponding predicate is true, such as in

```
(COND ((EQ X Y) (SETQ Y 5) (FOO X Y))
      (T (FIE Y)))
```

(EX 11)



In the foo-expression we know that y has value 5, but after the cond-expression we can only deduce that an assignment may have been performed and that a possible value for y is 5.

In

```
(COND ((NULL X) (FOO Y))
      ((EQ X (SETQ Y 5)) (FIE X Y))          (EX 12)
      (T (FUM Y)))
```

the assignment is done in a predicate of a clause and it holds that this assignment affects both the rest of the forms in the clause and the remaining clauses in the expression. After the cond-expression we can also here only deduce that a possible value of y is 5.

If, however, the assignment is done in the first predicate this assignment will hold for all clauses and also for the forms after the cond-expression, such as in

```
(COND ((EQ X (SETQ Z 5)) (FOO X Z))          (EX 13)
      (T (FIE X Z)))
```

The same case occurs also in and- and or-expressions in which the first form is always evaluated, such as in

```
(AND (EQ X (SETQ Y 5)) (FOO X Z))            (EX 14)
```

But it is not so simple that an assignment in the first form will always be performed. This form can of course also in its turn be a conditional, as

```
(AND (OR L (SETQ Y 5)) (FOO Y))              (EX 15)
```

If an assignment is done on a variable in every branch we know that the variable will be assigned when evaluated under the assumption that one predicate will always be true. An example is

```
(COND ((NULL X) (SETQ Y 1))
      ((EQ X Y) (SETQ Y 2))                  (EX 16)
      (T (SETQ Y 3)))
```

Other cases may also occur

```
(COND ((NULL X) (SETQ Y 1))
      ((EQ X Y) (SETQ Y 2))
      ((FOO (SETQ Y 3)) (FIE X Y))
      (T (FUM X Y)))
```

(EX 17)

In both these cases y will have either 1, 2 or 3 as value after the cond-expression has been evaluated if we assume that none of the functions foo, fie and fum performs any assignments of y.

6.7.2.1 Algorithm. The following algorithm is used to find those variables which will always be assigned in a cond-expression:

```
(COND (p1 e1)
      (p2 e2)
      .
      .
      .
      (pk)
      .
      .
      .
      (pn en))
```

Let varassigned(form) give those variables which will always be assigned in form.

a. Let y and x be empty sets. The set y is used for the requested variables and the set x is used to hold candidates for such variables.

b.  $v := \text{varassigned}(p_1)$   
 $x := \text{varassigned}(e_1)^*$

---

\* In the case where the clause is of type  $(p_k)$ , varassigned( $e_k$ ) will return the empty set.

c. Perform for  $i$  from 2 to  $n-1$

$v := v \cup (x \cap \text{varassigned}(p_i))$

if  $\neg (p_i \text{ is known always to be false})$   
     then  $v := v \cap \text{varassigned}(e_i)^*$

d. if  $(p_n \text{ is known always to be true})$

then  $v := v \cup (x \cap \text{varassigned}(e_n))^*$   
     else  $v := v \cup (x \cap \text{varassigned}(p_n))$

e.  $v$  contains the requested variables.

This algorithm works under the assumption that  $p_1$  is not always known to be false. This cannot appear in REDFUN-2 because such a predicate will be broken out from the cond-expression into a progn, as in the example

```
(COND ((SETQ X NIL))
      ((PUT A B NIL))
      ((FOO X) (SETQ Y T))
      ((SETQ Y NIL))
      (T (FUM X Y)))
```

(EX 18)

which is transferred to

```
(PROGN (SETQ X NIL)
      (PUT A B NIL)
      (COND ((FOO NIL) (SETQ Y T))
            ((SETQ Y NIL))
            (T (FUM NIL NIL))))
```

In the above example both  $x$  and  $y$  will always be assigned.

---

\* In the case where the clause is of type  $(p_k)$ ,  $\text{varassigned}(e_k)$  will return the empty set.

6.7.2.2 Setq's reducer. The place in REDFUN-2 where an assignment is discovered is in the setq's reducer-procedure. The result of the assignment will be stored in the assigninfo-element in the quoted-expression. The expression

```
(SETQ X 5)
```

will result in

```
(QUOTED (SETQ X 5)
        (:VALUE . 5)
        :SIDE
        NIL
        NIL
        ((X . (:VALUE . 5))))
```

(EX 19)

If the value(s) of the assigned variable is unknown it will be marked as NOBIN. This assignment information will then be transferred upwards to all super-expressions, i.e. every form which performs an assignment will contain that information in its quoted-expression. The expression

```
(FOO (CONS (SETQ X A) B) (SETQ Y NIL))
```

(EX 20)

will, if we assume foo not to perform any assignments and no information about a exists, result in

```
(QUOTED (FOO (CONS (SETQ X A) B) (SETQ Y NIL))
        NIL
        :SIDE
        NIL
        NIL
        ((X . NOBIN) (Y . (:VALUE . NIL))))
```

6.7.2.3 The :ADDVALUE-descriptor. In the cases where we cannot assume that an assignment will always be performed, such as in the conditionals, it must be marked in the assigninfo-element. Such a variable is marked by a value-descriptor of the form

```
(:ADDVALUE . value-descriptor)
```

The expression

```
(AND (FOO X (SETQ X 3)) (SETQ Y 'A))
```

will result in the following assigninfo-element

```
((X . (:VALUE . 3)) (Y . (:ADDVALUE . (:VALUE . A))))
```

If the variable a-list before this expression contains

```
((X . (:VALUE . 1)) (Y . (:VALUES . (V X))) ... )
```

it will after the and-expression be changed to

```
((X . (:VALUE . 3)) (Y . (:VALUES . (V X A))) ... )
```

For x a replacement of the value is performed, but for y an extension of its value range is made. This extension is performed through an or-reduction of the two value-descriptors.

6.7.2.4 The :SETQVALUE-descriptor. Consider the two examples

```
(COND ((AND (SETQ X Y) (EQ X 'A) (FOO L)) (FIE X))
      (T ... ))
```

and

```
(COND ((AND X (FOO (SETQ X NIL) L)) (FIE X))
      (T ... ))
```

During the processing of these two and-expressions the variable x will both be found to be assigned and it will be possible to extract property information about it which holds in the true-branch. When information is gathered from the quoted-expression of the elements in order to build up the quoted-expression for the whole and-expression it is important to take in consideration the order between the assignment of a variable and the form giving the latest truectxt-information about it.

If a variable was assigned to the value which is stored in the truectxt-element, that value-descriptor must be marked by a :SETQVALUE-descriptor and will appear as

```
(:SETQVALUE . value-descriptor)
```

The two examples will create following truectxt-element in the quoted-expression for the whole and-expression. The first example gives

```
((X . (:VALUE . A)))
```

and the second example

```
((X . (:SETQVALUE . (:VALUE . NIL))))
```

If an assignment will not always be performed, i.e. the case when the value-descriptor in the assigninfo-element is marked by :ADDVALUE (described in 6.7.2.3) it is marked in the same way in the truectxt-element. In the example

```
(COND ((AND X (OR Z (SETQ X T))) (FIE X)))
```

the and-expression will cause the truectxt-element

```
((X . (:ADDVALUE . (:VALUE . T))))
```

to be created.

6.7.2.5 Specialization, replacement and extension of a value-descriptor. The three different value-descriptors obtained from the previous examples in 6.7.2.4 will be treated in different ways when the new a-list is built up for the true-branch. They correspond to a specialization, a replacement and an extension of a value-descriptor.

We define a value-descriptor before the entry to an expression to be  $\alpha$  and the value-descriptor in the truectxt-element to be  $\beta$ . The new value-descriptor to be used in the true-branch will then be created by the following table.

operation	resulting value-descriptor
specialization	and-reduction of $\alpha$ and $\beta$
replacement	$\beta$
extension	or-reduction of $\alpha$ and $\beta$

A final example. Consider the following a-list

```
((X . (:VALUES . (A B C)))
 (Y . (:VALUES . (X Y Z)))
 (Z . (:VALUES . (1 2 3))))
```

and the expression

```
(AND (MEMB X '(B C D)) (SETQ Y 'W) (AND L (SETQ Z 4))) (EX 20)
```

The quoted-expression for this expression will be

```
(QUOTED (AND (MEMB X '(B C D))
              (SETQ Y 'W)
              (AND (SETQ Z 4)))
 (:VALUES . (NIL 4))
:SIDE
((X . (:VALUES . (B C D)))
 (Y . (:SETQVALUE . (:VALUE . W))) ← truectxt-element
 (Z . (:ADDVALUE . (:VALUE . 4))))
NIL
((Y . (:ADDVALUE . (:VALUE . W)))
 (Z . (:ADDVALUE . (:VALUE . 4))))
```

The value-descriptors for the variables x y and z to be used in the true-branch are calculated as follows.

For x a specialization is done of the previous value-descriptor and of the one in the truectxt-element by an and-reduction.

```
(:VALUES . (A B C)) ^ (:VALUES . (B C D)) → (:VALUES . (B C))
```

For y a replacement of the old value-descriptor is done and we obtain

```
(:VALUE . W)
```

For z an extension of the value-descriptor will be done through an or-reduction of the value-descriptor

```
(:VALUES . (1 2 3)) v (:VALUE . 4) → (:VALUES . (1 2 3 4))
```

6.7.3 Transferring of assignment information. The implementation of the transfer of assignment information is basically as follows. The setq-reducer discovers the assignment, as already has been described in 6.7.2.2. Transfer of assignment information from one argument to the next is done by redargs, which is simplified defined as

```
(LAMBDA (ARGS AL TEMP)
  (COND ((NULL ARGS) NIL)
        (T (CONS (SETQ TEMP (REDFORM (CAR ARGS) AL))
                  (REDARGS (CDR ARGS)
                           (ADD-ASSIGNINFO AL TEMP))))))
```

Transfer of the information from the arguments to the form is done for pure functions in tryapply, which in principle scans through the arguments and extracts the assigninfo-information. For reducer-functions this is done in their reducer-procedures.

6.7.4 Assignments by the function set. A function which can spoil all work here is set. If we receive a set-expression and have no knowledge about its first element, all information about variables must be removed. What must be done in such situations is that the user must supply information about which variables can be affected by this set.

Some cases can of course be handled by the set-reducer. The expression

```
(SET 'X 10)
```

is trivial. It will be collapsed and processed further by the setq-reducer. If the first argument performs a side-effect, as in

```
(SET (SETQ VAR 'X) 10)
```

the side-effect expression is broken out into

```
(PROGN (SETQ VAR 'X) (SET 'X 10)) (EX 21)
```

and the set-expression is further processed as above.



If the first argument to set is of values-type, such as in the example

```

.
.
.
(COND ((FOO X) (SETQ VAR 'A))
      ((FIE X) (SETQ VAR 'B))                (EX 22)
      (T (SETQ VAR 'C)))
(SET VAR 10)
.
.
.

```

we know that one of the variables a, b or c will be set to 10 and the assigninfo-element for the set-expression will be

```

((A . (:ADDVALUE . (:VALUE . 10)))
 (B . (:ADDVALUE . (:VALUE . 10)))
 (C . (:ADDVALUE . (:VALUE . 10))))

```

indicating that a possible value for these variables is 10.

If the first argument is of novalues-type, all variables on the a-list except those in the value-descriptor must be set to NOBIN. Nothing certain can be said about the variables in the remaining cases.

6.7.5 Global variables. In INTERLISP there are global variables. They can be assigned and accessed by rplaca\*(rpaq, rpaqg) resp. car\*, but will also be assigned or accessed by setq and by the variable itself respectively, if the variable is not bound on the parameter stack. The use of rplaca and car on globals is usually done for efficiency, the stack does not need to be searched every time the global variable is accessed, but also because the variable may have been used locally, i.e. bound as a lambda or prog-variable, and we really want to access its global value. The first case is not difficult it is only necessary to treat such variables as if they were bound in a global block regarding them as local variables. The other case can be more difficult to handle.

---

\* Follows the INTERLISP 360/370 implementation.

It is not always possible to know in a function if a variable will be used as a local or global one. The user can define that a variable shall always be considered as a global variable when ambiguity appears.

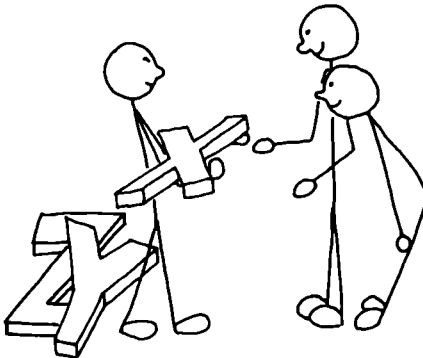
In REDFUN-2 the implementation is such that if the value is to be known for a global variable it is bound on the a-list. To distinguish it from locally bound values, the global variable var will be renamed as var-GLOBAL. The assignment and access will then be performed as follows:

(SETQ X ...)	}	If <u>x</u> is not defined as global use X, or if <u>x</u> is already bound on the a-list use X, otherwise use X-GLOBAL.
(SETQQ X ...)		
(SET 'X ...)		

(RPLACA 'X ...)	}	Use X-GLOBAL.
(RPAQ X ...)		
(RPAQQ X ...)		

(CAR 'X)	Use X-GLOBAL when searching the a-list.
----------	---

X (when evaluating the variable)	Search a-list first with X, if not stored and if <u>x</u> is defined as global search for X-GLOBAL.
----------------------------------	---



"VARIABLE ASSIGNMENT"

## 6.8 EXTENDED FUNCTION CLASS AUTHORITY.

A function will normally be classified as belonging to a function class. In some cases this function class must depend on the actual arguments of the function. For example car is a pure-function if its argument is a list, otherwise if it is an atom it belongs to another class and must be treated differently.

6.8.1 Function classes. Here follows a description of the various classes in REDFUN-2:

PURE           for a function, which does not perform or depend on any side-effects, and which can be evaluated at reduction-time if its arguments have known values.

For such a function redform will perform

```
tryapply[car[form],redargs[cdr[form],
                        al,rctxt],rctxt]
```

where tryapply was described earlier in 6.4.4 and form, al and rctxt are arguments to redform.

EXPR           for functions of expr- or fexpr-type (if the arguments will be evaluated in the normal order). The arguments will be reduced. Redform will perform

```
cons[car[form],redargs[cdr[form],al,rctxt]]
```

SIDEEXPR       for an eval-function\* performing side-effects. Redform will perform

```
sideexpr[car[form],redargs[cdr[form],
                           al,rctxt],rctxt]
```

where sideexpr in principle returns a q-tuple.

---

\* In INTERLISP either expr, expr\*, cexpr, cexpr\*, subr or subr\*.

```

<cons[fn,unquoted[args]],
                        apply[getp[fn,VALUEFN],args],
:SIDE,
NIL,
NIL,
collected-assigninfo-from-arguments>

```

FEXPR        for a noeval-function, which does not perform any side-effects. The form is returned unchanged.

SIDEFEXPR    for a noeval-function for which nothing is done for its arguments and a quoted-expression is returned which marks it to perform side-effects.

REDUCER      for a function normally with special argument evaluation. A reducer-property must then be associated with the function which performs the necessary reductions.

Redform will perform

```

apply[getp[car[form],REDUCER],cdr[form],rctxt].

```

OPEN         for a function one wants to open. This includes beta-expansion where a function call is replaced by the function body, open specialization, where the call is replaced by an open lambda- or proq-expression, and closed specialization, where a reduced version of the function is created and the call is replaced by this reduced version.

Reform will perform

```

expand[car[form],redargs[cdr[form],
                        al,rctxt]]

```

where expand decides what opening to perform, depending on either a sub-declaration for the function or an automatic procedure which decides what is most appropriate in this situation. The expression to insert instead of the call is returned from expand. This is discussed further in section 6.11.

6.8.2 Functions belonging to several classes. The function which performs this classification is extended not only to classify with respect to a function but also to the whole form. For those functions which can belong to several classes a fnclassfn-procedure must be associated with the function and that procedure can then decide to which class the form belongs. Some functions belong to different classes depending on the datatype of its arguments and in this class we have car, which if its argument is a list is treated as pure, but if it is an atom it depends on side-effects - the global assignment of variables. In that case car has a reducer-property. It may be desired to treat the function getp, which also depends on side-effects, in some situations as a pure-function if the property value is stored on the property list at reduction time, otherwise it should be treated as a sideexpr-function. Actually one can in REDFUN-2 declare those properties which will treat getp as a pure-function.

6.8.3 The function class of cons. A function which is really troublesome in this case is cons. It unquestionably perform side-effects, allocating a new cons-cell every time it is called. If we are not performing any side-effects on that structure created by cons, we can treat it as a pure-function and consequently evaluate it when its two arguments are known. If we make rplacd's, nconc's etc later on in the structure we may not be able to treat it as a pure. A good example of this, which appeared in our work, was together with the REMREC package. A recursive function will be translated there into a prog-expression and the values from the function will be stored in a queue.

This queue is built up by a header and the elements are nconc-ed into that queue. The header is set up by

```
(SETQ HEAD (CONS))  (*)
```

and later in the code values are put into it by

```
(NCONC1 HEAD ... )
```

Every time such a function is called we shall need a new header and a new queue, but if cons is treated as pure the REDFUN package will transform the \*-expression to

```
(SETQ HEAD (QUOTE (NIL)))
```

Which will have the consequence that every time we enter the function we shall start to build upon the old queue and an erroneous result will appear.

This problem is not only valid together with cons. It holds for all lists which will be treated as constants. If we have bound a constant list to a variable and later perform side-effects on it erroneous results can occur, shown by some examples

```
.
:
:
(SETQ X '(A B C))
(COND ((FOO Y) (NCONC X '(D E)) (FIE X))
      (T (FUM X)))
```

```
.
:
:
```

where the value of x in the truebranch will be changed to

```
(A B C D E)
```

and an erroneous result will occur if the a-list is not updated.

These changes in list structures are not so simple to detect, as in the example

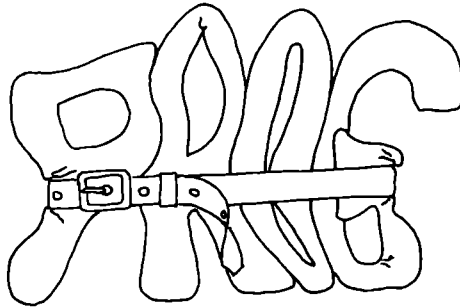
```
.  
.   
.   
(SETQ X '(A B C))  
(SETQ Y (CONS 'X (CDDR X)))  
(NCONC (CDR Y) '(D E))  
(FIE X)  
  
.   
.   
. 
```

A more careful analysis of side-effects performed on list-structures is desirable but has not been studied in this project so far.

## 6.9 REDUCTION OF PROG-EXPRESSIONS

A prog-expression reduces in principle by giving every statement to redform. If a reduced statement does not contain any side-effects it can be deleted. A clean-up of prog-variables no longer used, unnecessary assignments, goto's, labels etc are also desirable.

6.9.1 Assignments in a prog-expression. The main problem here occurs when assignments must be taken care of. REDFUN-2 is built up in such a way that information about assignments is collected in parallel with the reduction. The traversal in the code is done in the same order as the evaluator and when a form is processed all the information necessary to perform the reduction has already been collected. This works as long as we are not processing prog-statements. In a prog the evaluator can take several different paths which makes it difficult to collect all the information before the reduction takes place. If there are no loops in the prog it can be solved if one reduces the statements in such an order that all statements which can be evaluated before one arrives at a statement a are reduced completely before a is reduced. When loops are included an analysis must be performed to find those variables which can be assigned in the loop and for them only a limited amount of information can be found and used.



"PROG REDUCTION"



6.9.1.1 Prog-expressions without loops. Let us follow an example containing a prog without any loops.

```
(PROG (X Y)
  START
    .
    .
    (SETQ Y 5)
    .
    .
    (OR (FOO X) (GO B))
    .
    .
    (SETQ Y 10)
    .
    .
  A (PRINT Y)
    .
    .
    (RETURN ...)
  B (OR X (SETQ Y 7))
    .
    .
    (GO A))
```

We assume that these assignments are the only ones performed to the variable Y.

We can first illustrate the goto-structure better by a directed graph. The statements between two labels forms a block and is represented as the code in the code in the graph.

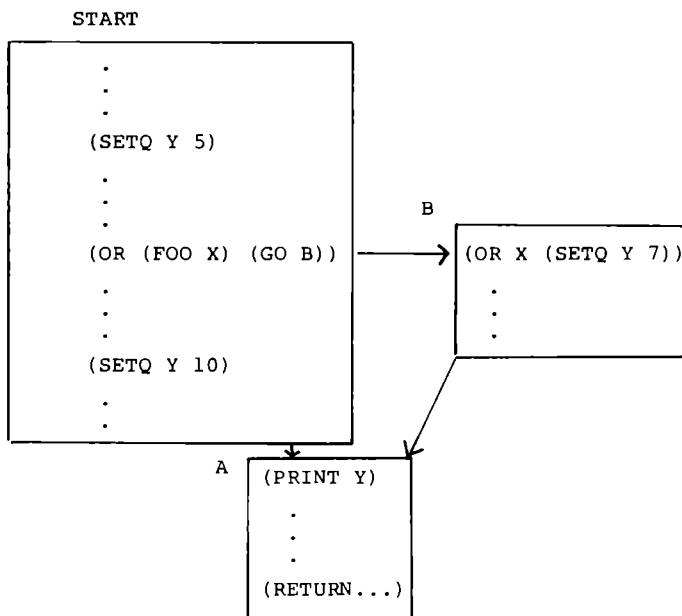


Fig 6.9.1

The first step is to build a table of this structure

label	come from
START	-
A	(START B)
B	(START)

A topological order of this graph gives us an order in which these blocks must be reduced and one such order is

START, B and A

When a go-statement or a label is reached during the reduction the information which holds about variables is associated with that label. When we proceed the reduction in a new block we collect that information. For each variable for which there is information we perform an or-reduction of its collected value-descriptors.

At label A in the example it holds that when coming from START

$$Y_{\text{values}} = \{10\}$$

and from B

$$Y_{\text{values}} = \{5 \ 7\}$$

and the or-reduction gives that

$$Y_{\text{values}} = \{5 \ 7 \ 10\}$$

holds at the label A.

6.9.1.2 Prog-expressions with loops. When one or several loops occur in the prog an analyses is performed to find those variables which are assigned inside a loop. Assume following directed graph representing a prog-expression.

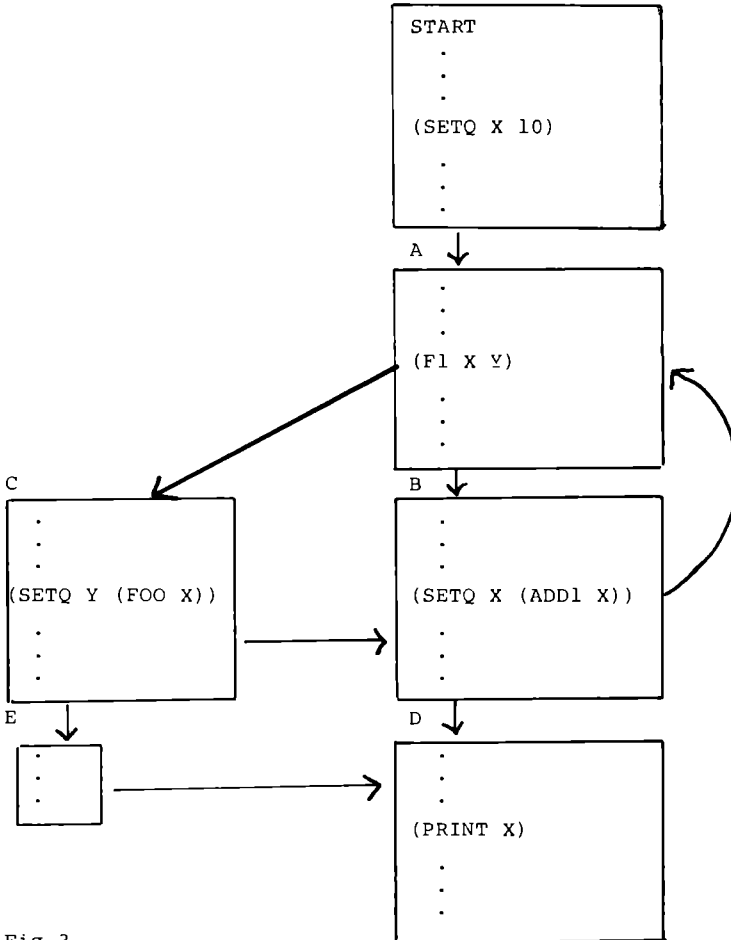


Fig 3.

The three blocks A, B and C form a loop. There are two variables  $\underline{x}$  and  $\underline{y}$  which may be assigned in that loop. When we encounter the block A during the reduction we must here assume these two variables to have unknown values.

Some information can be gathered about a variable in a loop and be used by the system at the reduction. If a variable is assigned to constant values or to values of a certain datatype that information can be used. Assume following graph

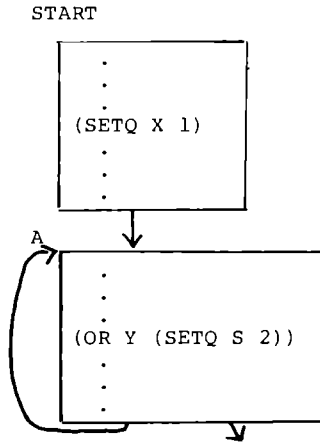


Fig 4.

We can here use the fact that  $\underline{x}$  has the value 1 or 2 at the entry to the block A.

The order in which the blocks are reduced is also here determined by a topological order, but all blocks in a loop are treated as one single block. The blocks inside a loop are then also ordered. The blocks in the example described in fig 3. are ordered as

START, A, C, B, E and D

6.9.1.3 Some problems occurring in this method. The analysis of the assignments found inside a loop is performed without using any knowledge about the variables. Parts of the code which will be eliminated can contain assignments which ought not to be included in the analysis. An example of such a case is

```
(PROG (X Y)
  .
  .
  .
  LOP
  .
  .
  .
  (COND ((EQ X T) (SETQ Y 1)))
  (COND ((NULL Y) (FIE X))
        (T (FUM X)))
  .
  .
  .
  (GO LOP)
  .
  .
  .
  )
```

We assume no other assignments to the variables x and y can occur. The analysis detects the loop and finds y to be a variable which may be assigned there. At reduction of the loop the variable y is assumed to have either NIL or 1 as value. The result after the reduction will then be

```
(PROG (X Y)
  .
  .
  .
  LOP
  .
  .
  .
  (COND ((NULL Y) (FIE NIL))
        (T (FUM NIL)))
  .
  .
  .
  (GO LOP)
  .
  .
  .
  )
```

The first cond-expression was eliminated and therefore also the assignment of the variable y.

The second cond-expression could be further reduced, now when we know that the only possible value for y was NIL. This problem can be solved in this case if we perform another analysis and reduction step in the prog-expression.

Another case which appeared in the application of REDFUN-2 to the iterative statement is described in section 7.4.3.3.

The same problem can also occur in the analysis of loops. Assume the example

```
(PROG ( ... )
  .
  .
  .
  LOP
  .
  .
  .
  (OR T (GO LOP))
  .
  .
  .
)
```

We assume that the only (GO LOP) appears in the or-reduction. The analysis will assume a loop here but in the reality we can never enter the loop. This can also be solved by performing one more analysis and reduction step.

6.9.2 Postprog-transformations. After the reduction of a prog-expression it is normally desired to perform a number of extra transformation steps of more or less cleaning-up nature, e.g. removing prog-variables not used in the prog, removing assignments to variables which will never be accessed, cleaning up the goto-structure etc.

There are lot of transformations which could be desirable to perform for a prog-expression, but different applications need different transformations. In the present version of REDFUN-2 some basic transformations are included and new transformations can be included when they are needed. Following transformations are included

- a. Find prog-variables which never will be accessed in the prog-expression or used as a free variable underneath. These variables can be deleted from the prog-variable list.
- b. Remove assignments to variables which are never accessed. Such assignments can remain after constant propagation, i.e. when a variable is assigned to a constant value and that constant has replaced all occurrences of the variable. If the form beeing assigned contains side-effects the form must be left in the code.
- c. Cleaning-up the goto-structure and removing unnessesary labels.

Other transformations of prog-expressions are also performed by collapser-rules. We can here mention the rule

$$(\text{PROG NIL } stm_1 \text{ } stm_2 \text{ } \dots \text{ } (\text{RETURN } stm_n)) \rightarrow (\text{PROGN } stm_1 \text{ } stm_2 \text{ } \dots \text{ } stm_n)$$

if no returns are performed in  $stm_i$ ,  $i=1,n$  and there is no gotos involved.

Some transformations and anlaysis can be made by programs automatically generated by PMG (RIS74). Around a skeleton, which in principle performs the code traversal, the user can specify to PMG what actions he wants to be performed at specified points in the code, for example an action performed for every variable encountered in the code, an extra operation for a certain function etc. This user supplied code is then integrated in the skeleton or put as "macros" and a complete program is generated. An advantage in using PMG-generated procedures is that they all follow the same conventions.



For functions with special argument structure these "macros" indicate how the traversal must be done. The user must define his own "macros" if such functions have to be processed in the code.

In REDFUN-2 the postprog-transformations a. and b. have been generated by the PMG-system.

## 6.10 OPENING OF FUNCTIONS

6.10.1 Open classes. Here we shall discuss both beta-expansion and specialization of functions. In the present system the following cases can occur

BETA	beta-expansion
LAMBDA	an open specialization by a <u>lambda-expression</u>
PROG	an open specialization by a <u>prog-expression</u>
REDUCED	a closed specialization

These are sub-declarations to the function class OPEN, and called open classes.

These concepts were defined in the section on REDFUN. New here is the PROG-case, which is only a variant of open specialization.

One can notice here that BETA and LAMBDA correspond to INTERLISP's substitution and open macros respectively for the compiler, although the substitution of the lambda-variables is done by the compiler without taken care of special functions with non-standard argument structure. The third macro variant for the INTERLISP compiler is the computed macro, which in principle corresponds to the code that REDCOMPILE produces.

6.10.2 Examples of open classes. Let us follow an example. Suppose foo is defined as

```
[LAMBDA (X Y Z)
  (COND [(ATOM X)
        (COND ((NULL Y)
               NIL)
              (T (FIE X Y Z))
              (T (FOO (CAR X)
                      (CAR Y)
                      Z))
        ])
```

### 6.10.2.1 Beta-expansion. A beta-expansion of

```
(FOO 'A (CAR L) NIL)
```

will result in

```
(COND ((NULL (CAR L))
      NIL)
      (T (FIE (QUOTE A)
              (CAR L)
              NIL)))
```

and can be done because x in foo is known to be an atom, so the recursive call to foo again has been eliminated. Nor were there any side-effects in the arguments to foo which could disturb the substitution. Another criteria for allowing beta-expansion is that the lambda-variables in foo are not used freely in any function called from foo, so we must assume fie not to use any variables freely in this example.

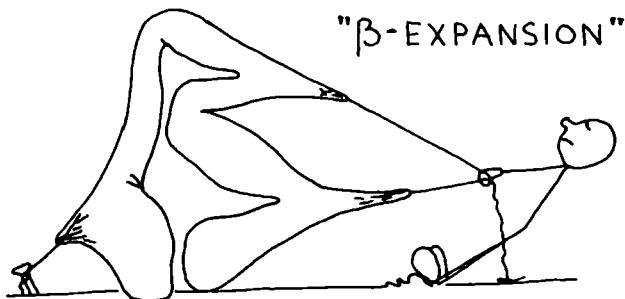
Although the first argument to foo is a list, but with known value, beta-expansion can be performed. The opening will then be made recursively.

```
(FOO '((X)) (CAR L) NIL)
```

will be replaced by

```
(COND (NULL (CAR (CAR (CAR L))
                NIL)
      (T (FIE (QUOTE X)
              (CAR (CAR (CAR L)))
              NIL)))
```

At beta-expansion the code can grow very quickly, especially in cases where the expansion is made recursively or on several levels.



### 6.10.2.2 Open specialization. For the expression

```
(FOO 'A (PRINT (CDR L)) NIL)
```

a beta-expansion is not appropriate. The form will if it is substituted in the body be evaluated twice and then per- from the print twice, which is not what one normally wants to be done. An open specialization solves this and the code would then be

```
([LAMBDA (Y)
  (COND ((NULL Y)
        NIL)
        (T (FIE (QUOTE A)
                  Y NIL]
    (PRINT (CDR L))))
```

or by a prog-expression

```
[PROG [(Y (PRINT (CDR L)
  (RETURN (COND ((NULL Y)
                  NIL)
                  (T (FIE (QUOTE A)
                           Y NIL]
    Y NIL]
```

It is assumed also here that fie does not use x or z freely. If that was the case these variables must be left in the variable list and the arguments to bind them must be left on the argument list.



"OPENING A FUNCTION"



The open class procedure makes the decision depending on the following information:

- a. side-effects among the arguments?
- b. is the function directly recursive, i.e. the call is done to itself in its function body or in a function which will be inserted in the body through beta-expansion?
- c. are there any assignments of the lambda-variables in the function body?
- d. are any of the lambda-variables used freely in functions calls from this one?

The desirable ordering is first beta-expansion, then open specialization and last closed specialization.

Case a. above is simple: the arguments have already been reduced an information about side-effects can be found in the q-tuple. For cases b. and c. one could go ahead and directly perform an analysis in the function body whether the function is recursive or not and check for assignments. The problem with such a scheme is that we include parts of the function body in our analysis, which may later be eliminated. This gives a worst case analysis. In the earlier example the function foo could in such a case only be opened to closed specialization. A slightly better analysis will be done if we first make a reduction of the function body with those lambda-variables whose values we know something about. Information about assignments will be collected automatically and a check whether the resulted body is recursive is simple to perform.

The procedure works as follows:

Suppose we have

$$\text{fn}[\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n]$$

which must be opened and the definition

$$\text{fn}[x_1, x_2, \dots, x_n] = \text{fnbody}$$

The arguments arg, have been reduced when entering this procedure.

- a. Will any of arg<sub>i</sub> perform a side-effect? In this case a beta-expansion will not be performed.
- b. Reduce fnbody through redform , with the a-list extended by those lambda-variables x<sub>i</sub> bound to its corresponding value-descriptor for arg<sub>i</sub>. If no value-descriptor is available the variable is bound to NOBIN. We shall receive a reduced body fnbody'.
- c. Check in fnbody' whether a recursive call is made to fn. If so, a closed specialization must be performed.
- d. Will any x<sub>i</sub> be assigned in fnbody'? If so, a beta-expansion cannot be done.
- e. Does any function called from fn use any x<sub>i</sub> as free variable? In this case a beta-expansion cannot be performed.

From this procedure and the desirable ordering an appropriate open class will be given.

This procedure will in some cases choose open specialization although beta-expansion could be used. When side-effects occur in the arguments a beta-expansion can be done if these forms are to be substituted in the body in such a way that when the body is evaluated the side-effected forms will only be evaluated once and in the right order with regard to other forms which depend on side-effects performed there. There are other cases too where the procedure makes a bad choice, but it seems to be rather complex to perform the analysis in order to cover such cases.

For beta-expansion the function body fnbody' must be reduced once more through redform, where the remaining lambda-variables are replaced by actual argument forms. This must be done to allow collapse rules to be invoked, which in its turn can cause further reduction to be done. The resulting body will be inserted in the code.

For open and closed specialization the function body fnbody' must be embedded in a lambda- or prog-expression, where those variables which were not affected in steps a, d or e in the above procedure could be removed.

6.10.4 Substitution package. A problem in REDFUN was that beta-expansion in some cases could be very ineffective. There were two problems. First, substitution was made in parts of the code which would later be eliminated. Some modifications there were done in order to let conditionals be processed more effectively. The second problem was that arguments which had just been reduced after substitution in the code were reduced again.

The first problem is solved if we can perform the substitution and the reduction in parallel. This is done in REDFUN-2 in such a way that the lambda-variables in the function to beta-expand are bound on the a-list to the forms which will be inserted there. There is a subst-descriptor which holds such a form and it is then redform's task to perform the substitution when the variable is encountered in the code. The whole substitution package can then be thrown away. A subst-descriptor looks like

```
(:SUBST . form)
```

The second problem is solved by the fact that every argument is enclosed in a quoted-expression and is not therefore reduced any further. By collapse rules however, one can force a reduction to be performed, which is necessary in some cases.



Some minor problems must be taken care of in this scheme. Consider the expression

```
(FOO X)
```

where x's value-descriptor is

```
(:VALUES . (1 2 NIL))
```

and foo defined as

```
(LAMBDA (L)
  (COND ((NUMBERP L) (ADD1 L))
        (T L)))
```

When the cond-expression is encountered at reduction the a-list contains

```
((L . (:SUBST . (QUOTED X (:VALUES . (1 2 NIL)))))
 (X . (:VALUES . (1 2 NIL)))
 .
 .
 . )
```

In numberp the variable l is substituted for x and information about x to be used in the true-branch tells us that x must be a number there. At add1 the a-list is

```
((X . (:VALUES . (1 2)))
 (L . (:SUBST . (QUOTED X (:VALUES . (1 2 NIL)))))
 (X . (:VALUES . (1 2 NIL)))
 .
 .
 . )
```

Here also l is substituted for x, but when a variable is inserted in the code a scan must be performed on the a-list to see if that variable has got another value-descriptor. In this case that descriptor must naturally be used instead, so add1 need only be evaluated for the values 1 and 2.

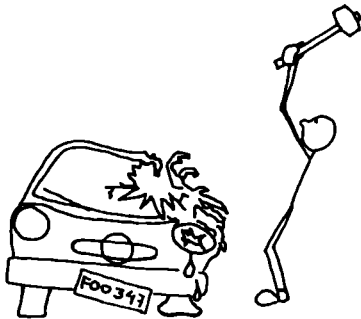
Another problem. Suppose we have the expression

```
(FOO (EQ X 'A) (FIE X))
```

and foo is defined as

```
(LAMBDA (B L)
  (COND (B L)
        (T ...)))
```

When b is substituted for the eq-expression it is in a position where a form should be reduced also to extract information for its true- and false-branch. No reduction will be performed after the substitution so that information must already have been extracted. This will only be done if a context reduction pattern\* to foo is given saying that its first argument must always extract such information.



"CAR COLLAPSER"

---

\* See section 6.12.5

### 6.11 COLLAPERS.

Some extensions of collpsers have been performed. In REDFUN a call to redform was always performed again after a collapsing rule had been invoked which caused efficiency problems and is solved in REDFUN-2 by the user explicitly performing such a call in the collapsing rule. This is done for the rule

```
(APPLY (QUOTE fn) (QUOTE arg1) ... (QUOTE argn)) →
      → (fn arg1 ... argn)
```

but not for

```
(CAR (CAR X)) → (CAAR X)
```

There is also a possibility of escaping from a collapsing rule by returning the atom NOCOLLAPS, which solves the problem discussed in 4.1.4.1. The expression

```
(APPLY* (QUOTE AND) X Y)
```

cannot be collapsed.

In some cases one must set restrictions when a collapsing rule can be used. The rule

```
(CAR (LIST X Y)) → X
```

can only be used under the assumption that Y does not perform any side-effects, in which case either the form is returned as it is or if one wants to avoid the conses an expression such as

```
(PROG1 X Y)
```

is used. However this introduces a problem in REDFUN-2. When the call to the collapser is made, all information about the sub-expressions in the form to be collapsed is lost.

## 6.12 REDUCTION IN CONTEXTS.

6.12.1 Reduction in contexts. The previous sections described how information is extracted from the forms being reduced. Such information includes the value(s) of the form and properties of variables to be used in the true- or false-branch, and is stored in a quoted-expression. Getting this information is in some cases rather time-consuming. It is therefore important to prevent this computation from being performed if we know that this information will never be used.

Examples.

- In a prog-expression the values of the statements are of no interest and need not be calculated.
- Consider

```
(COND ((MEMB X Y) (FOO X))
      (T (FIE X)))
```

and that x's and y's value-descriptors before the cond-expression are

```
(:VALUES . (A B C)) resp. (:VALUE . (A X C Y))
```

For the memb-expression we are here only interested whether it is true or false and not in the true-case what values made it true. The arguments to memb are known and will be classified to "allknownvalues" (see 6.4.4). All argument combinations must be calculated. Here we can interrupt that calculation when we have got at least one true and at least one false value, in which case we know that the memb-expression will not have the same boolean value and unnecessary computations are saved.

If however we perform

```
(SETQ Z (MEMB X Y))
```

the values are of interest and should be calculated if possible.

6.12.2 Contexts. Depending on its position in the code a form will be reduced with respect to a context. We have following contexts:

- value context, the value(s) of the form had to be calculated if possible
- boolean context, it is only needed to calculate if the value is true or false if possible.
- novalue context, the value had not to be calculated.

The extraction of variable properties is not useful to perform if the form is in such position in the code that these properties never can be used. For the value context and the boolean context we can also include there if such properties had to be extracted to be used in the true-branch, false-branch or both.

6.12.3 Context table. The different contexts can be represented in a table with abbreviated context names which will be used in the text.

	value	boolean value	no value
-	V	B	N
Properties about variables is to be extracted to be used in the <u>true-branch</u>	VT	BT	-
Properties about variables is to be extracted to be used in the <u>false-branch</u>	VF	BF	-
Properties about variables is to be extracted to be used both in the <u>true-branch</u> and the <u>false-branch</u>	VTF	BTf	-

Table 3. Context table

We shall first give examples of forms in these different contexts and then give formulas showing how the context changes for the arguments in conditionals, logical functions and some other functions.

6.12.4 Examples of a form in the different contexts:

VALUE (V)	The <u>cond</u> -expression in (PRINT (COND ((NULL L) 1) (T 2)))
NOVALUE (N)	The <u>cond</u> -expression in (PROG (X) . . . (COND ((NULL L) (SETQ X 1)) (T (SETQ X 2))) . . . ) )
BOOLEAN (B)	The <u>eq</u> -expression in (COND (... ...) ( (OR FLGA FLGB (EQ X Y)) ...))
VALUE + TRUE BRANCH (VT)	The variable <u>y</u> in (AND A B (SETQ X Y) ...)
VALUE + FALSE BRANCH (VF)	The <u>eq</u> -expression in (OR A B (EQ X 10) ...)
VALUE + TRUE and FALSE- BRANCH (VTF)	The variable <u>y</u> in (COND ((SETQ X Y) ...) (T ...))
BOOLEAN + TRUE BRANCH (BT)	The <u>eq</u> -expression in (AND A B (EQ X 'A) ....)

BOOLEAN + FALSE BRANCH (BF) The eg-expression in  
 (PROG (X)  
 .  
 .  
 .  
 (COND ((NULL L) (SETQ X 1))  
 ((EQ X 'A))  
 (T (FOO X)))  
 .  
 .  
 . )

BOOLEAN + TRUE and FALSE      The eg-expression in  
 BRANCH (BTF)                    (COND ((EQ X 'A) ...)  
                                   (T ...))



6.12.5 Changes of contexts. We shall here show how the context changes in some different functions. For the function progn it can be described as

$$\begin{aligned} (\text{PROGN } \alpha \dots \alpha \beta) \quad & \alpha = N \\ & \beta = \text{ctxt} \end{aligned}$$

which is interpreted as:

When a progn-expression is reduced in the context ctxt then all the arguments but the last one will be reduced in context N no value context and the last one in context ctxt. The interpretation of a capital in order to describe a context is found in table 3 in section 6.12.3. The context ctxt is used as the context in which the whole expression is reduced in.

For the similar functions prog1 and prog2 we get

$$\begin{aligned} (\text{PROG1 } \alpha \beta \dots \beta) \quad & \alpha = \text{ctxt} \\ & \beta = N \\ (\text{PROG2 } \alpha \beta \alpha \dots \alpha) \quad & \alpha = N \\ & \beta = \text{ctxt} \end{aligned}$$

Logical functions.

$$(\text{AND } \alpha_1 \alpha_2 \dots \alpha_{n-1} \beta) \quad \alpha_1 = \begin{cases} \text{BTF if } \text{ctxt} = \text{VT}, \text{VTF}, \text{BF} \\ \text{or BTF} \\ \text{BT in remaining cases} \end{cases}$$

$$\alpha_{i+1} = \begin{cases} \text{BTF if } \alpha_i = \text{BTF and } \text{falsectxtflg}_i = \underline{\text{true}} \\ \text{BT if } \alpha_i = \text{BT or } \text{falsectxtflg}_i = \underline{\text{false}} \end{cases}$$

$$\beta = \begin{cases} \begin{matrix} \text{V} \\ \text{VT} \\ \text{B} \\ \text{BT} \\ \text{ctxt in remaining cases} \end{matrix} & \text{if } \text{ctxt} = \begin{cases} \text{VF} \\ \text{VTF} \\ \text{BF} \\ \text{BTF} \end{cases} & \text{and } \text{falsectxtflg}_{n-1} = \underline{\text{false}} \end{cases}$$

To extract information for an argument to its falsectxt-element in an and-expression is only useful if such information is available for the same variable in all arguments. A flag falsectxtflg checks if there is at least one variable for which this holds and otherwise signals to change context. The flag is initialized to true.

Example

(AND L (EQ X 'A) (EQ Y 'B))

If the expression is reduced in a BTF context

L is reduced in BTF

(EQ X 'A) also in BTF

(EQ Y 'B) only in BT

This has been discussed earlier in section 6.5.5.1

$$\begin{aligned}
 & (OR \alpha_1 \alpha_2 \dots \alpha_{n-1} \beta) \\
 & \alpha_1 = \begin{cases} \text{VTF} \\ \text{VF} \\ \text{if ctxt} = \\ \text{BTF} \\ \text{BT} \end{cases} = \begin{cases} \text{VTF or VT} \\ \text{V or VF} \\ \text{BTF or BT} \\ \text{B,N or BF} \end{cases} \\
 & \alpha_{i+1} = \begin{cases} \alpha_i & \text{if truectxtflg}_{i-1} = \text{truectxtflg}_i \\ \text{VF} & \text{if ctxt} = \text{VTF} \\ \text{BF} & \text{if ctxt} = \text{BTF} \end{cases} \\
 & \beta = \begin{cases} \text{V} \\ \text{VF} \\ \text{if ctxt} = \\ \text{B} \\ \text{BF} \\ \text{ctxt in remaining cases} \end{cases} = \begin{cases} \text{VT} \\ \text{VTF} \\ \text{BT} \\ \text{BTF} \end{cases} \quad \text{and truectxtflg}_{n-1} = \underline{\text{false}}
 \end{aligned}$$

In the same way as for and in its false-branch, a flag truectxtflg is used to signal a context change for its true-branch.

(NOT  $\alpha$ )

$$\alpha = \begin{cases} B & \text{if } \text{ctxt} = V \text{ or } B \\ BF & \text{if } \text{ctxt} = VT \text{ or } BT \\ BT & \text{if } \text{ctxt} = VF \text{ or } BF \\ BTF & \text{if } \text{ctxt} = VTF \text{ or } BTF \\ N & \text{if } \text{ctxt} = N \end{cases}$$

Conditionals.(COND ( $\alpha_1$   $\beta$ ).  
.  
.( $\gamma_1$ ).  
.  
.( $\alpha_n$   $\beta$ )alt. ( $\gamma_n$ ) $\alpha_1 =$  BTF $\beta =$  ctxt

$$\gamma_i = \begin{cases} BF \\ VF \\ BTF \\ VTF \end{cases} \text{ if } \text{ctxt} = \begin{cases} B, BF \text{ or } N \\ V \text{ or } VF \\ BTF \text{ or } BT \\ VTF \text{ or } VT \end{cases}$$

$$\alpha_n = \begin{cases} BTF & \text{if } \text{ctxt} = VF, VTF, BF \text{ or } BTF \\ BT & \text{in remaining cases} \end{cases}$$

 $\gamma_n =$  ctxt(SELECTQ  $\alpha$ (-  $\beta$ )(-  $\beta$ ).  
.  
. $\beta$ ) $\alpha =$  V $\beta =$  ctxtProg-expression(PROG ( (-  $\beta$ ) ...) $\alpha$ 

.

 $\alpha$ (RETURN  $\gamma$ ) $\alpha$ 

)

 $\alpha =$  N $\beta =$  V $\gamma =$  ctxt

In cond, selectg and proq,  $\beta$  stands for the whole implicit progn-expression, so if there is more than one form in that position they are reduced in an N context with the exception of the last form.

### Assignment

$$(\text{SETQ} - \alpha) \quad \alpha = \begin{cases} V & \text{if } \text{ctxt} = V \text{ or } B \text{ or } N \\ VT & \text{if } \text{ctxt} = VT \text{ or } BT \\ VF & \text{if } \text{ctxt} = VF \text{ or } BF \\ VTF & \text{if } \text{ctxt} = VTF \text{ or } BTF \end{cases}$$

For other functions a context reduction pattern can be associated. It is simply a list describing the context each argument should be reduced in. This pattern contains either an abbreviated context name from the table 3 or a special element

\$RCTXT meaning the context the whole form is reduced in, \$VALUE as \$RCTXT, but if the whole form is reduced in a novalue-context it means V, or in a boolean context where the corresponding value context is chosen.

### Examples

PUT has pattern (N N \$RCTXT)  
 RPLACA has pattern (V \$VALUE). This pattern is used when the first argument is an atom, when a global assingment is done. The value of the second argument is then of interest.

By default a function will change context as

(FN  $\alpha$  ...  $\alpha$ )       $\alpha = V$

In appendix II is shown an example how these contexts changes in an expression.

## 7. A NEW APPLICATION AND NEW EXPERIMENTS WITH REDFUN-2

In this chapter we will first briefly discuss macros and especially macros in the INTERLISP system. The reader who is familiar with this can continue reading at section 7.3. In that section there is a discussion of how macros used by the INTERLISP compiler can automatically be generated and processed before compilation. The next section reports an experiment with the REDFUN-2 system on a fairly complex program. This program was written without any knowledge about our partial evaluation methods and gives a good idea about the complexity of code the REDFUN-2 system can manage.

### 7.1 A PARTIAL EVALUATOR AS A MACRO EXPANDER.

A macro in a programming language is normally a number of statements in that language, which will be substituted into the code instead of the macro call. Most assembly languages have macro facility and can also be found in high level languages. The macro expansion is normally performed in a pre-step before the actual assembling or compiling. A macro can normally be used with parameters and special macro or assembly variables are used to conditionally expand them.

As shown earlier in the report, opening of functions and beta-expansion is a kind of macro-expansion. By partial evaluation the macro can be expanded conditionally. It is interesting to notice here that a user of a macro needs to distinguish between macro variables used at expansion time and those variables used when executing the code. In partial evaluation the user need not really separate these two types of variable usage and this task is taken over by the partial evaluator.

In this new approach to viewing macros we can see every function or procedure as a macro candidate. Many implementations suffer today because a function or procedure call is rather expensive and the use of procedures is in many cases avoided for efficiency reasons. The use of procedures normally gives a more readable code, easier to maintain and debug. When the program goes into production simple procedures for which the overhead of performing a function call is high, can be treated as macros and expanded by the compiler or in a pre-step.

## 7.2 MACRO-EXPANSION IN THE INTERLISP SYSTEM.

Before compiling, a number of functions in INTERLISP are expanded through macros. Map-functions are expanded to prog-expressions and can therefore be more efficient when compiled. The user can also introduce his own macros for functions he has defined. In some LISP-systems there is also a macro facility whereby the code is expanded at evaluation time, but this will not be discussed here.

The macros in INTERLISP are of three kinds:

- open macros. The lambda body is inserted directly in the code and corresponds in REDFUN-2 to an open specialization.
- substitution macros. The lambda variables are substituted for the arguments in the function body which is then inserted in the code. This corresponds to beta-expansion.
- computed macros. The code to insert is calculated by the macro and corresponds to a function generated by REDCOMPILE (described in 4.1.5)

One can notice that the substitution in a substitution macro is simply done through subpair, i.e. all occurrences of the atom, which corresponds to a lambda-variable, replaced regardless of whether it acts as a variable or not. This can naturally introduce errors as in the following example.

Suppose

```
(LAMBDA (X) (SELECTQ X
                  (X 'A)
                  (Y 'B)
                  'C)))
```

is a definition of a substitution macro for foo. The expression

```
(FOO L)
```

will be expanded to

```
(SELECTQ L
  (L 'A)
  (Y 'B)
  'C)
```

where the substitution of the X in the first case statement in the selectq is erroneous.

The reason for using macros when compiling is obviously to remove function calls. For an open and substitution macro it is simple to give the function definition as a macro. One must be careful not to give recursive functions as macros. Side-effects and the way substitution is performed in the substitution macro also require caution. A computed macro is normally more complicated to write. It is important that the original function is consistent with the computed macro and when the original definition is changed the macro must be rewritten appropriately also.

### 7.3 DISCUSSION OF MAP-FUNCTIONS (APPLICATION H1)

Before compilation a map-function is expanded through a computed macro into a prog. If one looks at the macro definitions of a map-function it seems rather complex and it is somewhat difficult really to find out how it works. A number of errors in these definitions have occurred in different versions of INTERLISP. The macro has not been consistent with the parent functions definition. Erroneous code has been generated if the function which is applied to each element is of nlambda-type.

This has been the case both with INTERLISP/10 of the version used some years ago, INTERLISP/360-370 and also in the LISP compiler developed by Urm1 (URM77). Compilation of the function notany has also been erroneous on INTERLISP/360-370 if there was no third argument in the call. This shows the difficulty of writing a computed macro, which in all different cases which can appear, is consistent with the definition.

Our approach here is instead to generate the macros automatically. The steps could be as follows:

- a. Write a map-function with a recursive definition. This should be the only version to be maintained.
- b. Run this recursive version through a recursion remover, e.g. REMREC (RIS73) and an iterative version is generated. This version can be compiled and used as the system function
- c. During compilation of a map-expression, a partial evaluation of the iterative version in LISP format with regard to the arguments can be performed. The result could then be compiled.

Partial evaluation in step c. can seem a little inefficient to perform every time and a redcompilation of the iterative version in LISP format is desirable.

Using this scheme there is only one version to maintain, the recursive version, and all other versions are generated automatically. The only problem which remains is of course to be sure that the various program generators generate correct programs.

One can go further. Instead of defining all these different map-functions one could write one or a number of more general maproutines and then by partial evaluation generate a recursive version for each of these basic ones.



Appendix III shows some examples of these different steps. The function mapcar was defined recursively and the various versions were generated and processed. The other map-functions follow the same scheme but have not been run through these steps. The REDCOMPILE system could not properly generate code for the prog-expressions but with some help it worked in this test.

#### 7.4 EXPERIMENTS WITH THE ITERATIVE STATEMENT (APPLICATION H2)

7.4.1 Description of the application. The iterative statement, based on the one in CLISP (TEI74 section 23) has been implemented as a LISP program by Jim Goodwin during his stay at Linköping University. The program can handle a large number of variants of such statements as.

```
(FOR I FROM 1 TO 10 DO (PRINT I))
(FOR I FROM 1 TO N BY 2 SUM I)
(WHILE (LESSP I 5) DO (FOO I) (SETQ I (SUB1 I)))
```

and a more fancy one as

```
(FOR X IN L BIND Y FIRST (SETQ Y 0) DO
      (IF (ATOM X) THEN (SETQ Y (ADD1 Y)))
      FINALLY (RETURN Y))
```

which returns the number of atoms in a list.

The program is implemented in two steps.

1. The iterative statement is parsed and during the parsing 32 variables are bound to values which fully describe the statement.
2. A prog-expression is set up in order to bind local and loop variables from the statement. It contains only a call to the function which executes the statement. All the variables set up in step 1 are then free variables in this executor function. The prog-expression is then evaluated and the iterative statement performed.

This implementation of the iterative statement differs from the one in CLISP, where the statement is parsed directly into LISP code. This translating is only performed once, the first time the iterative statement is encountered in the code during evaluation. A problem with this solution is that one must keep two different versions of the statements. The translated version should be invisible to the user and therefore the editor, prettyprint routines, file package etc. must be able to handle these two versions correctly.

We shall not discuss these two different approaches but only establish the fact that we have the package written by Jim Goodwin and see what can be done with it. In an interpreted environment one can let the parsing and the execution remain as it is, although it can seem a little inefficient to perform the parsing every time the statement is evaluated. It is more serious when the iterative statement has to be compiled. In such code it is unsatisfactory to parse and execute the statement in this manner. One line of approach is to write a macro for the statement and we come back to the problem that we have different versions of the program to maintain.

The method we use is instead to let the program remain as it is as the only version to be maintained. At compiling time an iterative statement will be processed as follows. Perform the parsing and create the simple prog-expression. Beta-expand the call to the executor and its sub-functions and partially evaluate. Hopefully the result will be a specialized version of the executor which corresponds to this iterative statement and which can be efficiently compiled.

These experiments are also a good test of REDFUN-2, especially since the program was written by an expert LISP programmer, and without any knowledge of REDFUN-2 or our partial evaluation ideas.

7.4.2 Performance. The parsing step is left aside here and the point of interest is the executor functions. Appendix IV shows the LISP code for these functions and illustrates the complexity of the code we are operating on.

These examples has been run through the system succesfully. The problems which occurred could easily be fixed and they are discussed in section 7.4.3. The iterative statement is rather complex in itself and covers many possibilities and it is difficult to say if REDFUN-2 has processed all the different cases which can appear, and in a satisfactory way.

We start by showing the results from the first two examples given before.

Example 1

```
(FOR I FROM 1 TO 10 DO (PRINT I))
```

Reduced code from REDFUN-2:

```
(PROG (RANGE:LEFT I)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  $$LP
  (SELECTQ (GREATERP RANGE:LEFT 10)
    (T (GO $$OUT))
    NIL)
  (PRINT I)
  (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
  (GO $$LP)
  $$OUT
  (RETURN NIL))
```

## Example 2

```
(FOR I FROM 1 TO N BY 2 SUM I)
```

Reduced code from REDFUN-2:

```
(PROG (RANGE:LEFT TO:LEFT I ITEMP ITER:VALUE)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  (SETQ TO:LEFT N)
  (SETQ ITER:VALUE 0)
  $$LP
  (SELECTQ (GREATERP RANGE:LEFT TO:LEFT)
    (T (GO $$OUT))
    'NIL)
  (SETQ ITEMP I)
  (SETQ ITER:VALUE (PLUS ITER:VALUE ITEMP))
  (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 2)))
  (GO $$LP)
  $$OUT
  (RETURN ITER:VALUE))
```

We may clarify some details of how the iterative statement works in this implementation. We can see from the examples that there are two loop variables, the external one given in the statement as in this case i, and an internal variable range:left. Both are increased in every step, the external one is used when the loop body is executed and the internal one only to control the loop. The upper bound for the loop is bound (in ex 2) to a variable to:left, which is used when checking the exit condition. This scheme prohibits the user from changing the loop-conditions in the loop, although the external variable i can be changed.

On these assumptions the code in these examples is nearly as good as one can write it.

7.4.3 Detected problems. We shall now discuss some problems which arose, which cannot be generally solved by REDFUN-2 and the actions taken to overcome these problems. These are items I; in the figure in section 3 of the thesis.

#### 7.4.3.1 Unrolling of prog-expressions.

##### a. The problem.

Among the subfunctions for the executor there was one function, segmap, which was written as a prog-expression. The function was defined as

```
[LAMBDA (**L **R FN)
  (PROG (**V)
    **LP(COND
      ((EQ **L **R)
        (RETURN **V))
      ((NLISTP **L)
        (ITEROR SEGMAP)))
    (SETQ **V (APPLY* FN (CAR **L)))
    (SETQ **L (CDR **L))
    (GO **LP])
```

A typical call to the function is

```
(SEGMAP '((PRINT I) (SETQ I (ADD1 I))) NIL (FUNCTION EVAL))
```

Actually, this is the function which executes the body in the iterative statement.

This function should be opened, but REDFUN-2 can only perform an open specialization and leave the prog-expression as it is. The first argument, however, is a constant list and it would therefore be desirable to perform unrolling of the prog-expression and receive

```
(PROGN (PRINT I) (SETQ I (ADD1 I)))
```

when expressions like

```
(APPLY* (FUNCTION EVAL) '(PRINT I))
```

have been collapsed to

```
(PRINT I)
```

b. Analysis of the problem.

To unroll a loop is to repeat the statements in the loop sequentially and to perform appropriate substitutions of the loop-variable, described by the example

```
.
.
.
(SETQ I 5)
LOP
(COND ((EQ I 7) (GO ON)))
(PRINT I)
(FOO I)

(SETQ I (ADD1 I))
(GO LOP)
ON
.
.
.
```

can be unrolled to

```
.
.
.
(PRINT 5)
(FOO 5)
(PRINT 6)
(FOO 6)
.
.
.
```

In arbitrary loop-structures unrolling is a difficult task to perform, if it is possible at all. The difficulty is to identify the loop variables and the actions performed in every iteration of the loop. For map-functions or for-expressions, such as in the iterative statement, it is simpler to do this because the loop variable is explicitly given in the statement or is implicitly given through the implementation (for map-functions). Naturally one could have an analysis program which could detect simple structures which directly correspond to a for-loop or the like, and perform unrolling for them.

### c. Temporary solution.

This problem can be solved if we define segmap recursively, simply as

```
[LAMBDA (**L **R FN **V)
  (COND
    ((EQ **L **R)
     **V)
    ((NLISTP **L)
     (ITEROR SEGMAP))
    (T (SEGMAP (CDR **L)
               **R FN (APPLY* FN (CAR **L))
```

If we define segmap now to belong to the function-class OPEN and open-class BETA, REDFUN-2 will recursively beta-expand it and the desired result will be achieved.

This caused, however, another problem which is discussed next.

Remark. This was the only change performed in the original code in order to be able to process it by REDFUN-2. The author of segmap has further remarked that the function was written iteratively in the first place mainly because the existing compiler did not handle recursion efficiently, i.e. lacked recursion-to-prog conversion.

### 7.4.3.2 Beta-expansion.

#### a. The problem

The following case arose when the function segmap was recursively beta-expanded. When a recursive call to segmap is beta-expanded, the fourth argument will never appear in the resulting code if another recursive expansion follows. This happens because the variable \*\*v does not exist in the third branch of the cond-expression. Unfortunately, if there is a side-effect in this fourth argument, the side-effect disappears also, incorrectly.

b. Analysis of the problem

At beta-expansion, arguments performing side-effects, and never inserted in the resulting code, must remain.

If foo is defined

```
(LAMBDA (X Y)
  (COND ((ZEROP X) NIL)
        (T Y)))
```

the expression

```
(FOO 0 (SETQ Z I))
```

if beta-expanded, must result in

```
(PROGN (SETQ Z I) NIL)
```

c. Solution

REDFUN-2 was updated to handle this case.

7.4.3.3 Interrelation between reduction and analysis of code.

As long as we are not processing code containing loops, the reduction and the analysis of assignments can proceed in parallel. When loops are included the analysis and the reduction must be performed in separate steps.

a. Problem

A case which appeared in this application was the following (simplified)

```
(PROG ((BY:TEST 'GREATERP) ...)
  LOP
  ... (SELECTQ BY:TEST
        (GREATERP (GREATERP RANGE:LEFT TO:LEFT))
        (COND
          ( ... )
          (T ... )))
  :
  :
  (AND (NUMBERP BY:TEST) (SETQ BY:TEST 1))
  (GO LOP)
  :
  :
  )
```



Before the loop the variable by:test is bound to GREATERP. The analysis of assignments will find that by:test can be assigned to 1 and therefore REDFUN-2 will assume that the two possible values for by:test in the loop are GREATERP and 1. The assignment of by:test to 1 can never be performed if the initial value of by:test is GREATERP! Therefore we could exclude this as a possible value.

#### b. Analysis of the problem

This problem has earlier been discussed in section 6.9.1.3. This case, however, cannot be solved by performing another step of analysis of assignments and reduction. This depends on the fact that the variable by:test also occurs in the condition which controls the assignment of that variable. The and-ex-expression will therefore never be eliminated. We need a reasoning based on the induction principle to be able to conclude that the variable by:test never can be assigned to a number.

#### c. Temporary solution

The assignment of the variable by:test is performed in a sub-function to the executor which will be beta-expanded. This assignment appears only in a very special use of the iterative statement. In this test example we excluded that variable from the list of variables which may be assigned in that sub-function.

### 7.4.3.4 Set-expressions

#### a. Problem

A set-expression was encountered during the analysis of assignments. This set-expression assigns the external loop-variable.

#### b. Analysis of the problem

We must know what values the first argument to set can take, i.e. which variables may be assigned by that expression. If the first argument varies in the loop this can be a problem. As shown earlier it is difficult to extract information about a variable in a loop, and here we must extract such information

on two levels.

### c. Temporary solution

We assumed that the first argument in the set-expression would not change value in the loop. When that expression is encountered during the analysis we can simply look up in the a-list and see if it has a known value. In our application, this variable was always bound to the external loop-variable so there were no other problems.

7.4.4 Other amendments. In order to make the examples go through successfully it was necessary to give some additional information and transformations to REDFUN-2.

#### a. A collapser

(OR X NIL) → X

which of course should be included as a standard collapser.

#### b. The information that the functions greaterp, lessp, zerop and listp only can return T or NIL values (declared on the property list of these functions).

A remark here. A problem in a system like REDFUN-2 is that it is difficult to know what information is really needed in the system. One question was whether it was necessary to know that these "predicate" functions, including eq and many others, could only return T or NIL as values. One would expect that these functions would only be in a predicate context and that we are therefore only interested in whether the value is true or false. We decided not to include this information in the system. In the first real experiment there was in fact a need for it. The code arising after simplification was:

```
(SELECTQ (GREATERP RANGE:LEFT 10)
  (T (GO $$OUT))
  (SKIP (GO $$ITERATE))
  NIL)
```

which obviously should be reduced to

```
(SELECTQ (GREATERP RANGE:LETF 10)
  (T (GO $$OUT))
  NIL)
```

c. Postprog-transformations performed.

- c1. To find those variables in a prog, which will never be accessed, and remove them from the prog-variable list.
- c2. Remove the assignments to those variables found in step C1.
- c3. Remove initializations of variables in the prog-variable list if the variable will also be assigned before the first label in the prog-expression\*

Example

```
(PROG ((I 1) X)
  (SETQ I 2)
  LOP
  :
  .)
```

is transformed to

```
(PROG (I X)
  (SETQ I 2)
  LOP
  :
  .)
```

- c4. Remove labels not used in the prog-expression.

---

\* This transformation will only be performed if the variable is not used before the assignment and is not applicable on the example

```
(PROG ((I (FOO J)))
  (PRINT I)
  (SETQ I 1)
  LOP
  :
  .)
```



## 8. CONCLUSIONS

This chapter will evaluate the REDFUN-2 program and summarize some conclusions and experiences gained from this work and propose some directions for further development of the REDFUN-project.

### 8.1 SUMMARY OF THIS THESIS

This thesis has described the progress of the REDFUN-project. An analysis of a former version and gained experiences from experiments with that version led us to propose a new version, the REDFUN-2 program. The background for this work, the new design, the implementation and experiments have been reported in the chapters 3 to 7. We will in this section evaluate the REDFUN-2 program and summarize some conclusions and experiences made so far found during the work with the system.

8.1.1 The REDFUN-2 program. REDFUN-2 is a quite large program. It consists of about 400 LISP functions and takes about 7000 lines (120 printout pages) to print in a prettyprinted format. It occupies 60000 words (120 memory pages) in the INTERLISP/20 system on a DEC-20. For a comparison, the first version of REDFUN (program B in fig 1) consisted of about 1000 lines prettyprinted code and REDFUN' (version E) of about 1500 lines. This new version is consequently about 5-7 times larger than its forerunners. If the next version grows in the same speed as this version it will be a fairly large program. The LISP coded part of INTERLISP/360-370 is prettyprinted on about 300 pages and the INTERLISP/20 system must be 2-3 times larger.

8.1.2 The feasibility of the proposed design. The design which was the basis for the REDFUN-2 implementation was described in section 5.2. The design has in most cases been sufficient in order to implement the new features and extensions described in section 5.1. The g-tuple has contained enough information with information from the code needed to perform appropriate reductions and by the semantic procedures it has been easy to include relevant properties about LISP-functions needed during the reduction. The design, however, is not sufficient to handle all cases which can appear in prog-expressions (when arbitrary goto's are performed). This does not cause REDFUN-2 to perform erroneous transformations, but the analysis will in some cases not be as good as we would like, and it will cause that some reductions are never performed. Some problems have already been reported in section 7.4.3. These problems, however, were of such nature that they were caused by the absence of features which the system was not intended to manage. Next section will show some other cases where the design was not appropriate.

### 8.1.3 Some weaknesses in the proposed design.

8.1.3.1 Handling of go-statements. It is necessary to keep track of go-expressions in the same manner as assignments. A new element in the g-tuple is desirable in order to handle the go-information in an expression. It is necessary to know whether from an expression we can continue to the next one or whether we always leave the expression by a branch (unless we are not branching to a label following the expression). Consider the example

```
(PROGN
  (COND ((NULL X) (GO A))
        ((EQ X 'A) (GO B))
        (T (GO C)))
  (FOO X))
```

We can not continue from the cond-expression to the following foo-expression and that expression can therefore be eliminated

from the code. It is also important during the loop analysis to consider such cases. The same problem occurs also when return-expressions are involved. A solution is to mark in the q-tuple to which labels we may branch and if we can pass through the expression without performing a go. The only reduction of dead code in a prog-expression is performed between a top level (unconditional) go-statement and the following label, such as in the example

```
(PROG (X)
  .
  .
  .
  (GO A)
  (FOO X)
  A
  .
  .
  .
  ))
```

and also after a top level return-statement.

The go-statement can also effect the handling of assignments. In the example

```
(COND ((NULL X) (GO A))
      ((EQ X 'A) (SETQ X 1))
      (T (SETQ X 2)))
```

an assignment of the variable x has always been performed when the following expression is encountered. In our system this case will be treated as if an assignment of x may have been performed.

8.1.3.2 Handling of side-effects. In the q-tuple we only flag (with a true or false value) whether a side-effect may occur in an expression or not. We must, however, distinguish between different kinds of side-effects, especially whether an assignment in a prog-expression is performed to a local variable or a global one. Inside the prog-expression the assignment must be treated as a side-effect but outside it should not be treated as such. When the prog-reducer in our system checks

for side-effects it cannot find out from the q-tuples of the reduced statements, if there are only side-effects on local variables or not. In the first case the whole prog-expression should not be marked to perform a side-effect and in the second case it should be marked.

#### 8.1.3.3 Problems in collapsers and postprog-transformations.

When a form has been processed and its q-tuple is created, all information is forgotten about the arguments in the form. Sometimes when we perform posttransformations on an expression we would like to extract that information again. Consider the example

```
(PROG (X Y Z)
      (SETQ X 10)
      (SETQ Z (PUT A B 'C))
      (FOO X (SETQ Y 5))
      (RETURN (FIE X Y Z)))
```

We assume foo and fie not to perform any side-effects. When this expression has been reduced we can apply the postprog-transformation which remove variables (and assignments of such variables) which are never accessed in the code. When such assignments are removed the second argument in the setq-expression must remain in the code if the assignment is performed in a value-context or if the second argument performs side-effects. This information, however, must be re-computed and cannot in our system be extracted from the q-tuple. Another solution is to let the second argument remain in the code and perform another reduction step.

A more general solution is to let all information about a form remain. Such information can be associated to the code through hash-arrays. The risk with this scheme is naturally that we will get an explosion of information saved about a program and we may probables run into space problems.



8.1.4 Complexity. Is REDFUN-2 complex? The answer is yes. A goal when implementing the system was to structure the program in a good way. In many cases we have succeeded; reducers and semantic procedures are examples of that.

There is, however, another complexity involved in the program. A large number of transformations, analysis, property extractions from the code etc go on in large portions of the program. A reducer is responsible for the reduction of an expression and to collect and process all necessary information from its arguments q-tuple in order to create the new q-tuple. Further, at every place where reduction of code is performed this transfer and processing of information from q-tuples must be performed. A user which writes a reducer for a function must also know about all this processing and include it there.

Some complexity can be removed if reductions, evaluation order of arguments in special functions, handling of assignment information etc could be described in a more high-level notation. Reductions and simplifications can for example be described by production rules.

8.1.5 Generality. Is REDFUN-2 general? The answer is yes, relatively general, if one means that REDFUN-2 can operate on arbitrarily written code and not be limited to a subset of LISP or to a very "pure" version of it. Naturally there are limitations in the code we can process, especially in a language as LISP in which in principle everything is allowed. The experiment with the iterative statement is a good example of its generality, where the code was written by an expert LISP programmer and was written without any knowledge about our partial evaluation ideas and the REDFUN-programs.

The answer is no, if one means that we should be able to perform all kinds of transformations by the system. The purpose with this system is to manipulate programs where the main operation is based on partial evaluation, and we have included

some other operations which seem to be useful in connection with that method.

8.1.6 Reliability. Is REDFUN-2 reliable? The answer is no. There is a real problem to manipulate LISP code where the user is allowed to do more or less what he wants. The statement

```
(EVAL (READ))
```

spoils all analysis and knowledge we have about the program, if there is no information about what statements can be read in and evaluated by that expression. As described in section 6.7.4 the function set can spoil all information we have about variables in program if we do not know which variables may be assigned by that set-expression. The functions rplaca and rplacd are also difficult to handle, and were briefly discussed in 6.8.3. The reliability requirement is however not as strong when the system is used interactively, and the user can watch what is going on in the system, and the system can ask for additional information from the user during processing. The reliability requirement is much more important if the system acts without any supervision. This is the case if the system is incorporated in a LISP compiler to take care of the macro-expansion which was discussed in chapter 7. One solution could be if we had a program which could analyse other LISP-programs and classify its code, whether it is purely written or written with "uncontrollable" side-effects. Depending on such analysis the program can then decide to which degree the manipulation can be allowed to be performed.

8.1.7 Efficiency. Is REDFUN-2 efficient? Answer: not too bad. In the implementation we have not spent too much effort to perform a very efficient implementation. One goal, however, was that when we chose between features to include in this new version we only included such that could be implemented in such a way that it could be used reasonably efficient.

We wanted to be able to run quite large examples through the system and not only "toy-problems". To reduce an iterative statement, where the code to reduce consisted of about 100 lines prettyprinted code (see appendix IV) takes about 20 seconds. Time in INTERLISP/20. As comparison, to compile these executor functions takes about 8 seconds.

## 8.2 DIRECTIONS FOR FURTHER WORK

### 8.2.1 The REDFUN-2 program.

- Find further test examples and applications. To use the REDFUN-2 program as a macro expander seems promising and further experiments ought to be performed in that application. Another application is to "compile" an interpretation over a constant datastructure. Such an example is found in a system which interprets conversation graphs (HAG76). A conversation between a user and a program is represented as a graph. An interpreter for the graph is then available in order to perform the conversation. The advantage of this representation is that it is simple to experiment with the conversation and changes in the conversation causes simple updates in the graph. In a production system it can seem inefficient to interpret the conversation and it would be desirable to "compile" the graph into a program. This "compilation" can under certain assumptions be performed through partial evaluation.
- Extend REDFUN-2's ability to handle more complex knowledge about the program. Such a knowledge can be
  - "x is greater than 10"
  - "l is a list of 3 elements"
  - "the third element in the list l is the atom A"
 or more complex expressions and include a theorem prover to prove assertions from them.
- Include new features in the system. New features will normally arise when new experiments are performed. Examples which can be desirable are to take care of common sub-expressions, extend the analysis and processing of prog-expressions and optimize code through rearranging of code.

- Combine the system with other program analysis and manipulation programs. A number of analysis routines in REDFUN-2 for example finding free variables, can be better performed by specialized analysis programs, such as FUNSTRUC (NOR72) and MASINTERSCOPE (TEI74) in the INTERLISP system. Conventions how different programs communicate with each other and other similar problems must be solved.
- If REDFUN-2 is combined with other systems it would also be desirable to design a database structure to store information about the program being processed. Such a database should also contain information about different versions of the program being processed and know which transformations have been performed to it.

#### 8.2.2 The REDCOMPILE program.

- The REDCOMPILE program was described in 4.1.5 and used in an example in appendix III. In many applications the use of REDFUN-2 can seem too inefficient. In some cases it can be solved if the program being partial evaluated was redcompiled. If the executor procedures in the implementation of the iterative statement could be redcompiled we would have been able to generate a computed macro which can be used when such a statement is compiled. This program can also be compared with the routine in CLISP which translates an iterative statement to LISP code. The REDCOMPILE-program needs to go through another iteration step and to be able to process the same complexity of code that REDFUN-2 can do.

If such a new version of REDCOMPILE could be made it would probably be more practically useful than the corresponding REDFUN-program.

- In the REDFUN-report we discussed that REDCOMPILE applied to a program P in principle performs the same operation as if REDFUN was applied to itself with the program P as the constant information (given as the second argument to REDFUN, i.e. the a-list in the present system). In order to achieve this, REDFUN must be sufficiently powerful to accept itself as argument and at the same time be written in such a way that it is an acceptable argument to itself. Try to achieve this goal. To perform this is probably more interesting in principle than practically useful.

### 8.2.3 Theoretical work

- Our approach in program manipulation has been to develop useful programs and use them in some applications and from that develop methods and gain more knowledge of problems which occur. The complementary approach, to develop a theory of program manipulation, and from that develop the practical methods, is of course also very important. We have earlier (in section 2.6) mentioned some important projects in this field. It is important when times arrives to derive results from such work and put it in practical use by intergrating them in our systems. Such an important result is to have an analysis program which determines where in the code optimizations are necessary. A problem today is that we spent much time in parts of the code which are executed non-frequently and where optimizations are of no use. If it significant to those parts -tight loops, frequent function calls, etc -where all resources in optimization should be spent. Interesting results in this direction have been reported by Wegbreit (WEG75b).







## APPENDIX I

## GENERATING STOREDEF-PROCEDURES IN THE PCDB APPLICATION

Here follows a more detailed description, illustrating how the reduction is performed in the PCDB-examples in section 4.1.2.1, where functions were beta-expanded through several levels.

The lambda-expression in the funarg-expressions, from which the reduction proceeds, is the same for the three exemplified relations. The values bound to the funarg-variables will differ.

```

<LAMBDA (A B)
  (COND
    <(ONEONE ONE)
      (PRG (ROOT)
        (RETURN (COND
          ((TESTER P A B LOC
            (CAR TYP)
            (CADR TYP))
            ROOT)
          ((STOREP (REV P)
            B A (QUOTE ONE)
            LOC
            (CADR TYP))
            (PLACA ROOT B)
            (T (APPLY* (FILLAND ONE)
              (QUOTE (STOREP P A B (CADR ONE)
                LOC
                (CAR TYP)))
              (QUOTE (STOREP (REV P)
                B A (CAR ONE)
                LOC
                (CADR TYP))

```

The definitions of the functions involved in the reduction steps are shown here again (they were also shown in section 4.1.2.1)

The functions tester, storer, getter and comparer are declared open and will be beta-expanded and the functions oneone, filland and fillfunc are pure. Their definitions are

STORER

```
<LAMBDA (R A B N L TI)
  (APPLY* (FILLFUNC N)
    (GETTER R A L TI)
    B)
```

TESTER

```
<LAMBDA (P A B LOC TA TB)
  (AND (CAP (SETQ ROOT (GETTER P A LOC TA)))
    (PROG2 (SETQ ROOT
      (APPLY* (COMPARER (QUOTE ONE)
        TB)
        (CAP ROOT)
        B))
    T)
```

GETTER

```
<LAMBDA (P A L TI)
  (OLDCOND
    ((AND (MEMB (QUOTE AA)
      TI)
      (AA A))
    (SELECTQ L
      (ARGS (GETROOT A P))
      (PRED (PGETROOT (GETROOT
        P
        (QUOTE TRUEFOR))
        A))
      ((HE CYCYCHE)
        (GETROOT A P))
      NIL))
    ((AND (MEMB (QUOTE HX)
      TI)
      (HX A))
    (SELECTQ L
      (ARGS (GETROOT (CAP A)
        P))
      (PRED (PGETROOT (GETROOT
        R
        (QUOTE TRUEFOR))
        (CAP A)))
      ((HE CYCYCHE)
        (GETROOT (CAP A)
        P))
      NIL))
    ((MEMB (QUOTE SX)
      TI)
    (PGETROOT (GETROOT R (QUOTE TRUEFOR))
      A)
```

COMPARER

```

<LAMBDA (N TI)
  (COND
    ((MEMB (QUOTE SX)
      TI)
      (SELECTQ N
        (MANY (FUNCTION MEMBER))
        (ONE (FUNCTION EQUAL))
        NIL))
    (T (SELECTQ N
      (ONE (FUNCTION EQ))
      (MANY (FUNCTION MEMB))
      NIL)
  )

```

ONEONE

```

<LAMBDA (ONE)
  (EQUAL ONE (QUOTE (ONE ONE)

```

FILLAND

```

<LAMBDA (ONE)
  (COND
    ((EQUAL ONE (QUOTE (ONE MANY)))
      (FUNCTION REVAND))
    (T (FUNCTION AND)
  )

```

FILLFUNC

```

<LAMBDA (NC)
  (SELECTQ NC
    (ONE (FUNCTION FILLROOT))
    (MANY (FUNCTION ADDROOT))
    NIL)

```

The CHILD relation.

Values of the free variables

R - CHILD  
 ONE - (ONE MANY)  
 LOC - ARGS  
 TYP - ((AA) (AA))

Reduction steps:

- a. Evaluation of oneone to NIL and filland to (FUNCTION REVAND) and an invocation of the collapser rule

```
(APPLY* (FUNCTION fn)
  (QUOTE arg1) (QUOTE arg2) ... (QUOTE argn)) →
→ (fn arg1 arg2 ... argn)
```

which holds if fn is of noeval-type, will result in

```
<LAMBDA (A B)
  (REVAND (STORED (QUOTE CHILD)
    A B (QUOTE MANY)
    (QUOTE ARGS)
    (QUOTE (AA)))
  (STORED (QUOTE REVCHILD)
    B A (QUOTE ONE)
    (QUOTE ARGS)
    (QUOTE (AA))
```

- b. Beta-expansion of storer

```
<LAMBDA (A B)
  (REVAND (ADDRoot (GETTER (QUOTE CHILD)
    A
    (QUOTE ARGS)
    (QUOTE (AA)))
    B)
  (FILLROOT (GETTER (QUOTE REVCHILD)
    B
    (QUOTE ARGS)
    (QUOTE (AA)))
    A>
```

The functions addroot and fillroot are results from evaluation of fillfunc with argument MANY resp. ONE. A collapse rule

```
(APPLY* (FUNCTION fn) arg1 arg2 ... argn) →
→ (fn arg1 arg2 ... argn)
```

which holds if fn is of eval-type has also been invoked.

c. Beta-expansion of getter gives

```
<LAMBDA (A B)
  (REVAND (ADDROOT (GETROOT A (QUOTE CHILD))
                  B)
    (FILLROOT (GETROOT B (QUOTE REVCHILD))
              A>
```

d. A collapse rule

```
(REVAND arg1 arg2) → (AND arg2 arg1)
```

gives the final code

```
<LAMBDA (A B)
  (AND (FILLROOT (GETROOT B (QUOTE REVCHILD))
                 A)
    (ADDROOT (GETROOT A (QUOTE CHILD))
             B>
```

### The MARRIED relation

Values of free variables

```
R    - MARRIED
ONE  - (ONE ONE)
LOC  - ARGS
TYP  - ((AA) (AA))
```

Reduction steps:

a. Evaluation of oneone to true gives

```

<LAMBDA (A B)
  (PROG (ROOT)
    (RETURN (COND
      ((TESTER (QUOTE MARRIED)
        A B (QUOTE ARGS)
        (QUOTE (AA))
        (QUOTE (AA)))
      ROOT)
      ((STOREP (QUOTE REVMARRIED)
        B A (QUOTE ONE)
        (QUOTE ARGS)
        (QUOTE (AA)))
      (PLACA ROOT B)
    )
  )

```

b. Beta-expansion of tester and storer and the use of the collapse rule given in step b for the CHILD relation gives

```

<LAMBDA (A R)
  (PROG (ROOT)
    (RETURN
      (COND
        ((AND (CAP (SETQ ROOT
          (GETTER (QUOTE MARRIED)
            A
            (QUOTE ARGS)
            (QUOTE (AA))
          (PROG2 (SETQ ROOT
            (EQ (CAP ROOT)
              R))
            T))
          ROOT)
          ((FILLROOT (GETTER (QUOTE REVMARRIED)
            R
            (QUOTE ARGS)
            (QUOTE (AA)))
            A)
          (PLACA ROOT B)
        )
      )
    )

```

c. Beta-expansion of getter results in the final code

```

<LAMBDA (A B)
  (PROG (ROOT)
    (RETURN
      (COND
        ((AND <CAP (SETQ ROOT
          (GETROOT A (QUOTE MARRIED)
        (PROG2 (SETQ ROOT
          (EQ (CAP ROOT)
            B))
          T))
        ROOT)
        ((FILLROOT (GETROOT B (QUOTE REVMARRIED)
          )
          A)
        (RPLACA ROOT B)

```

### The AGE relation

Values of the free variables:

```

R    - AGE
ONE  - (ONE MANY)
LOC  - ARGS
TYP  - ((AA) (AA))

```

Reduction steps:

Follows exactly the same steps as the CHILD relation and we obtain

```

a.  <LAMBDA (A B)
      (REVAND (STORES (QUOTE AGE)
        A B (QUOTE MANY)
        (QUOTE ARGS)
        (QUOTE (AA)))
      (STORES (QUOTE REVAGE)
        B A (QUOTE ONE)
        (QUOTE ARGS)
        (QUOTE (EX)

```

- b. <LAMBDA (A B)  
 (REVAND (ADDPROT (GETTER (QUOTE AGE)  
           A  
           (QUOTE ARGS)  
           (QUOTE (AA))))  
       B)  
 (FILLROOT (GETTER (QUOTE REVAGE)  
           B  
           (QUOTE ARGS)  
           (QUOTE (SX))))  
 A>
- c. <LAMBDA (A B)  
 (REVAND (ADDPROT (GETROOT A (QUOTE AGE))  
           B)  
 (FILLROOT (RGETROOT (GETROOT  
                       (QUOTE REVAGE)  
                       (QUOTE TRUEFOR))  
           B)  
 A>
- d. <LAMBDA (A B)  
 (AND (FILLROOT (RGETROOT (GETROOT (QUOTE REVAGE)  
                                   (QUOTE TRUEFOR))  
           B)  
       A)  
 (ADDPROT (GETROOT A (QUOTE AGE))  
 B>



## APPENDIX II

## CHANGES OF CONTEXTS - AN EXAMPLE

Here is shown an example how the context is changed in an expression. The interpretation of contexts and the context table are found in section 6.12.

The trace shows a sub-expression and the context it will be reduced in.

\*\*\*\*\*

```
FORM=
CAND (COND ((AND (EQ X (QUOTE A))
                  L
                  (SETQ Y W))
            (FOO X))
      ((OR (EQ X (QUOTE B))
            Z)))
      (OR (SETQ Z X)
          (PROGN (SETQ Y V)
                 (FIE Y))
```

AL=

CONTEXT=V

```
FORM= (AND (COND ((AND (EQ X (QUOTE A)) L (SETQ Y W))
                    (FOO X)) ((OR (EQ X (QUOTE B)) Z)))
        (OR (SETQ Z X) (PROGN
                    (SETQ Y V) (FIE Y))))
CONTEXT= V
```

```
FORM= (COND ((AND (EQ X (QUOTE A)) L (SETQ Y W)) (FOO
X)) ((OR (EQ X (QUOTE B)) Z)))
CONTEXT= BT
```

```
FORM= (AND (EQ X (QUOTE A)) L (SETQ Y W))
CONTEXT= BTF
```

```
FORM= (EQ X (QUOTE A))
CONTEXT= BTF
```

```
FORM= X
CONTEXT= V
```

```
FORM= (QUOTE A)
CONTEXT= V
```

```
FORM= L
CONTEXT= BTF
```

```
FORM= (SETQ Y W)
CONTEXT= BT
```

```
FORM= W
CONTEXT= VT
```

```
FORM= (FOO X)
CONTEXT= BT
```

FORM= X  
CONTEXT= BT

FORM= (OR (EQ X (QUOTE B)) Z)  
CONTEXT= BT

FORM= (EQ X (QUOTE B))  
CONTEXT= BTF

FORM= X  
CONTEXT= V

FORM= (QUOTE B)  
CONTEXT= V

FORM= Z  
CONTEXT= BT

FORM= (OR (SETQ Z X) (PROGN (SETQ Y V) (FIE Y)))  
CONTEXT= V

FORM= (SETQ Z X)  
CONTEXT= VF

FORM= X  
CONTEXT= VF

FORM= (PROGN (SETQ Y V) (FIE Y))  
CONTEXT= V

FORM= (SETQ Y V)  
CONTEXT= N

FORM= V  
CONTEXT= V

FORM= (FIE Y)  
CONTEXT= V

FORM= Y  
CONTEXT= V

REDUCED FORM=  
[AND (COND ((AND (EQ X (QUOTE A))  
                  L  
                  (SETQ Y W))  
              (FOO (QUOTE A)))  
      ((OR (EQ X (QUOTE B))  
           Z)))  
      (OR (SETQ Z X)  
          (PROGN (SETQ Y V)  
                  (FIE Y))

SIDE-EFFECT=YES

ASSIGN-INFO=  
  (Z :ADDDVALUE , NOBIN)  
  (Y :ADDDVALUE , NOBIN)

\*\*\*\*\*



## APPENDIX III

## MAP-FUNCTIONS - EXAMPLES

We will here show some examples how map-functions can be treated, in order to facilitate the maintainance of the various versions, a map-function can appear in. The principle idea was to start with a simple definition, easy to write and understand, and then by automatic transformations generate all other versions. The macro-expansion can then be performed through partial evaluation.

We take mapcar as example. A simple definition is naturally the recursive one

```
(MAPCAR
  (LAMBDA (L FN1 FN2)
    (COND
      ((NLISTP L)
       NIL)
      (T (CONS (APPLY* FN1 (CAR L))          (1)
                (MAPCAR (COND
                        (FN2 (APPLY* FN2 L))
                        (T (CDR L)))
                        FN1 FN2]))))
```

This definition is straightforward and should be the only version to maintain.

---

\* More unique names of the lambda-variables would be desirable but these ones are used for readability.

As system function an iterative version would probably be more efficient and a recursion remover can generate such version. We use the REMREC-program (RIS73) and receive

```

[LAMBDA (L FN1 FN2)
  (PROG (A0009 A0008)
    (SETQ A0008 (SETQ A0009 (CONS NIL NIL)))
    MAPCAR
      (COND
        ((NLISTP L)
          (RPLACD A0009 NIL)
          (RETURN (CDR A0008)))
        (T (RPLACD A0009 (CONS (APPLY* FN1 (CAR L))
                                NIL))
          (SETQ A0009 (CDR A0009))
          [SETQ L (COND
            (FN2 (APPLY* FN2 L))
            (T (CDR L)
              (GO MAPCAR))
          ]
        )
      )
  )

```

This version can be compiled and also used as the base from which we will perform partial evaluation.

#### Example I

Suppose that we at compile time encounter the expression

```
(MAPCAR NLIST (FUNCTION ADD1))
```

We will then open mapcar and from version 2 generate a specialized version suitable to compile. By REDFUN-2 we will get

```

[PROG ((L NLIST)
  A0009 A0008)
  (SETQ A0008 (SETQ A0009 (CONS NIL NIL)))
  MAPCAR
    (COND
      ((NLISTP L)
        (RPLACD A0009 NIL)
        (RETURN (CDR A0008)))
      (T (RPLACD A0009 (CONS (ADD1 (CAR L))
                              NIL))
        (SETQ A0009 (CDR A0009))
        (SETQ L (CDR L))
        (GO MAPCAR)
      )
    )

```

Here an open specialization has been performed with a prog-expression (mapcar was defined to belong to function class OPEN and open class PROG). A beta-expansion cannot be performed here because the variable l will be assigned in the prog-expression. Actually we got

```
(PROG ((L NLIST))
      (RETURN (PROG (A0009 A 0008)
                    ... )))
```

which was collapsed into the inner prog with its variable-list extended with the variables from the outer prog\*. Otherwise there was no real problems here. The apply\*-collapser is invoked and with fn2 known as NIL the inner cond-expression will be reduced.

Example II

```
(MAPCAR BREAKFNS (FUNCTION BREAK) (FUNCTION CDDR))
```

gives

```
[PROG ((L BREAKFNS)
      A0009 A0008)
  (SETQ A0008 (SETQ A0009 (CONS NIL NIL)))
  MAPCAR
  (COND
    ((NLISTP L)                                     (4)
     (RPLACD A0009 NIL)
     (RETURN (CDR A0008)))
    (T (RPLACD A0009 (CONS (APPLY* (QUOTE BREAK)
                                   (CAR L))
                           NIL))
      (SETQ A0009 (CDR A0009))
      (SETQ L (CDR L))
      (GO MAPCAR])
```

---

\* This is not allowed if any prog-variable in the inner prog is initialized by a form using a prog-variable from the outer one. Example

```
(PROG ((L 10)) (RETURN (PROG ((X L)) ...)))
```

cannot be transformed to

```
(PROG ((L 10) (X L)) ...)
```

## Example III

```
(MAPCAR NLIST (F/L (X) (FOO X)) TAILFN)
```

gives

```
[PROG ((L NLIST)
      (FN2 TAILFN)
      A0009 A0008)
  (SETQ A0008 (SETQ A0009 (CONS NIL NIL)))
  MAPCAR
    (COND
      ((NLISTP L)
       (RPLACD A0009 NIL)
       (RETURN (CDR A0008)))
      (T (RPLACD A0009 (CONS ([LAMBDA (X)
                              (FOO X)]
                              (CAR L))
                              NIL)))
       (SETQ A0009 (CDR A0009))
       (SETQ L (COND
         (FN2 (APPLY* FN2 L))
         (T (CDR L))
       )
       (GO MAPCAR])
    )
  )
(5)
```

Another kind of expansion of map-functions is when the first argument is known or is known how to compute. Instead of performing a loop over the elements in the list we want to produce straight code, where successive calls to the function are performed for each element. This can be done if we perform the partial evaluation and beta-expansion with the recursive definition as base. \*

The example

```
(MAPCAR (LIST I J K) (FUNCTION ADD1))
```

will be expanded to

```
(CONS (ADD1 I)
      (CONS (ADD1 J)
            (CONS (ADD1 K)
                  NIL))))
(6)
```

---

\* Compare the discussion about unrolling of the segmap-function in section 7.4.3.1.



Another example

```
(MAPCAR (LIST I J K L M N) (FUNCTION ADD1) (FUNCTION CDDR))
```

is expanded to

```
(CONS (ADD1 I)
      (CONS (ADD1 K)
            (CONS (ADD1 M)
                  (CONS (ADD1 N)
                        NIL))))
```

(7)

To perform this expansions we need following collapse rules:

(CDR (LIST x1 x2 ... xn))	→	(LIST x2 ... xn)
(CDR (LIST x))	→	NIL
(CDDR (LIST x1 x2 ... xn))	→	(LIST x3 ... xn)
(CDDR (LIST x1 x2))	→	NIL
(CAR (LIST x1 x2 ... xn))	→	x1
(NLISTP (LIST x1 x2 ... xn))	→	NIL

A computed macro will generate the code to compile more efficiently than we can perform through partial evaluation, so the next step is to generate such version of mapcar. The REDCOMPILE program could not completely perform this, but with some guidance from us, it worked. REDCOMPILE can only operate on programs, which are purely beta-expanded and not as in this case open specialized.

Version 2 of mapcar was run through REDCOMPILE and following code was generated:

```

[LAMBDA
  (L FN1 FN2)
  (LISTPROG
    (QUOTE PROG)
    (LIST (QUOTE A0009)
          (QUOTE A0008)
          (LIST (QUOTE L)
                L))
    [LIST (QUOTE SETQ)
          (QUOTE A0008)
          (LIST (QUOTE SETQ)
                (QUOTE A0009)
                (LIST (QUOTE CONS)
                      (KWOTE NIL)
                      (KWOTE NIL)
                )
          )
    (QUOTE MAPCAR)
    (LISTCOND
      (QUOTE COND)
      [LIST/NIL (LIST (QUOTE NLISTP)
                     (QUOTE L))
              (LIST (QUOTE RPLACD)
                    (QUOTE A0009)
                    (KWOTE NIL))
              (LIST (QUOTE RETURN)
                    (LIST (QUOTE CDR)
                          (QUOTE A0008)
                    )
              )
      )
    (LIST/NIL
      (KWOTE T)
      (LIST (QUOTE RPLACD)
            (QUOTE A0009)
            (LIST (QUOTE CONS)
                  [COLLAPS
                    (LIST (QUOTE AFFLY*)
                          FN1
                          (LIST (QUOTE CAR)
                                (QUOTE LJ)
                              )
                        )
                    (KWOTE NIL)))
            )
      (LIST (QUOTE SETQ)
            (QUOTE A0009)
            (LIST (QUOTE CDR)
                  (QUOTE A0009)))
      [LIST (QUOTE SETQ)
            (QUOTE L)
            (LISTCOND
              (QUOTE COND)
              [LIST/NIL FN2
                (COLLAPS (LIST (QUOTE AFFLY*)
                              FN2
                              (QUOTE LJ)
                            )
                )
              (LIST/NIL (KWOTE T)
                        (LIST (QUOTE CDR)
                              (QUOTE LJ)
                            )
                        )
              )
            )
      (LIST (QUOTE GO)
            (QUOTE MAPCAR)
      )
    )
  )

```

In the code the functions listprog, listcond and list/nil work as list, but perform also simple reductions. For example

```
listcond[COND, NIL, ((LISTP X) (FOO X)), (T (FIE X))] =
      (COND ((LISTP X) (FOO X)) (T (FIE X)))
```

and

```
listcond[COND, NIL, (T (FOO X))] =
      (FOO X)
```

To take care of the apply\*-expressions. REDCOMPILE was told to insert calls to the collapser-part (the function collaps) in REDFUN-2 and simplify such expressions at generation time.

By calling apply with the version (8) and the argument list of a mapcar-form the prog-expression to compile is generated. This version (8) will generate identical code for the examples I and II, and in example III tailfn is directly substituted into the code instead of fn2.

As comparison we finally give the LISP code for the definition and the macro definition of mapcar. They should be compared with versions (2) and (8). But notice that these both versions had to be maintained in the INTERLISP system for mapcar, but in our scheme only version (1). The definitions are taken from INTERLISP/10 (from 1972).

The interpretative definition of mapcar:

```
(MAPCAR
  [LAMBDA (MAPX MAPFN1 MAPFN2)
    (PROG (MAPL MAPE)
      LP (COND
        ((NLISTP MAPX)                                     (9)
         (RETURN MAPL)))
        (SETQ MAPE (CONS (APPLY* MAPFN1 (CAR MAPX))
                          MAPE))
        (COND
          (MAPL (FRPLACD (CDR MAPE)
                        (FRPLACD MAPE)))
          (T (SETQ MAPL MAPE)))
        (T (SETQ MAPX (COND
          (MAPFN2 (APPLY* MAPFN2 MAPX))
          (T (CDR MAPX))
          (GO LP))
```

This definition and the one automatic generated (2) differ very little. Version (2) performs one extra cons compared with this definition which performs one extra test inside the loop.

The macro definition:

```

[EX
  (PROG (LL Q)
    (RETURN
      (SUBPAIR
        (QUOTE (MAPX MAPCF MAPCF2 B))
        (LIST
          (CAR X)
          [COND
            [(SETQ Q (CFNP (CADR X)))
              (CONS Q (QUOTE ((CAR MACROX]
                (T [SETQ LL (CONS (LIST (QUOTE MACROF)
                                      (CADR X]
                  (QUOTE (APPLY* MACROF (CAR MACROX]
                [COND
                  [(CDDR X)
                    (COND
                      [(SETQ Q (CFNP (CADDR X)))
                        (CONS Q (QUOTE (MACROX]
                        (T (SETQ LL
                            (CONS (LIST (QUOTE MACROF2)
                                      (CADDR X))
                            LL))
                        (QUOTE (APPLY* MACROF2 MACROX]
                    (T (QUOTE (CDR MACROX]
                    LL)
                  (QUOTE
                    (PROG ((MACROX MAPX)
                      MACROY MACROZ MACROW . B)
                    MAPCLP
                      (COND
                        ((NLISTP MACROX)
                          (RETURN MACROY)))
                        (SETQ MACROW MAPCF)
                      [COND
                        [MACROZ
                          (SETQ MACROZ
                            (CDR
                              (FRPLACD MACROZ
                                (FRPLACD
                                  (CONS MACROW
                                    MACROZ]
                                (T (SETQ MACROY (SETQ MACROZ
                                    (CONS MACROW]
                                  (SETQ MACROX MAPCF2)
                                  (GO MAPCLP]

```

This variable x is bound to the argumentlist in the mapcar-form. The function cfnp is used to check if

```
(fn (CAR MACROX))          (a)
```

or

```
(APPLY* fn (CAR MACROX))   (b)
```

had to be built in order to perform the application of the function of each element in the list to map over. This corresponds to our apply\*-collapser.

The same examples I to III are also shown here. Notice that (4') is erroneous. The (a)-format above was chosen instead of the (b)-format\*.

```
(PROG ((MACROX NLIST)
      MACROY MACROZ MACROW)
  MAPCLP
    (COND
      ((NLISTP MACROX)
       (RETURN MACROY)))
      (SETQ MACROW (ADD1 (CAR MACROX)))
    )
  )
  (COND
    [MACROZ (SETQ MACROZ
                  (CDR (FRPLACD MACROZ
                                (FRPLACD (CONS MACROW
                                                MACROZ))
                                )
                  )
    ]
    (T (SETQ MACROY (SETQ MACROZ (CONS MACROW]
      (SETQ MACROX (CDR MACROX))
      (GO MAPCLP))

(PROG ((MACROX BREAKFNS)
      MACROY MACROZ MACROW)
  MAPCLP
    (COND
      ((NLISTP MACROX)
       (RETURN MACROY)))
      (SETQ MACROW (BREAK (CAR MACROX)))
    )
  )
  (COND
    [MACROZ (SETQ MACROZ
                  (CDR (FRPLACD MACROZ
                                (FRPLACD (CONS MACROW
                                                MACROZ]
                                )
                  )
    ]
    (T (SETQ MACROY (SETQ MACROZ (CONS MACROW]
      (SETQ MACROX (CDDR MACROX))
      (GO MAPCLP))

(4')
```

---

\* In INTERLISP/20 (from 1977) it is corrected

```

(PROG [(MACROX NLIST)
      MACROY MACROZ MACROW (MACROF2 TAILFN)
      (MACROF (F/L (X)
                (FOO X])
MAPCLP
(COND ((NLISTP MACROX)
      (RETURN MACROY))) (5')
(SETQ MACROW (APPLY* MACROF (CAR MACROX)))
[COND [MACROZ
      (SETQ MACROZ
        (CDR (FRPLACD MACROZ
                      (FRPLACD
                        (CONS MACROW MACROZ])
        (T (SETQ MACROY (SETQ MACROZ (CONS MACROW]
(SETQ MACROX (APPLY* MACROF2 MACROX))
(GO MAPCLP))

```





## APPENDIX IV

## THE EXPERIMENT WITH THE ITERATIVE STATEMENT

We will in this appendix give the full LISP code for the executor functions in the iterative statement implementation (described in 7.4). We will also give some output from various runs performed during the experiment.

We will first give a sampler of different iterative statements which can be executed by these functions:

```
(FOR I FROM 1 TO 10 DO (PRINT I))

(FOR I FROM 1 TO N BY 2 SUM I)

(NEVER (ATOM X) FOR X IN LLIST)

(IN AL COLLECT CAR)

(FOR X IN L BIND Y FIRST (SETQ Y 0)
 DO (COND ((ATOM X) (SETQ Y (ADD1 Y))))
 FINALLY (RETURN Y))

(FOR X IN L JOIN (CDR X) WHEN (AND (LISTP X)
                                   (EQ (CAR X) 'A)))
```

These examples are run through REDFUN-2 and are discussed in this appendix. For further description of the variants of the iterative statement see its documentation in the INTERLISP manual (TEI74).





## ITERUPDATE

-----  
[LAMBDA

NIL

(SELECTQ RANGE:OF [IN (SET FOR:LEFT (CAR (SETQ RANGE:LEFT  
(

ITERUPDATE1])

[(ON GENERATOR)

(SET FOR:LEFT (SETQ RANGE:LEFT (ITERUPDATE1])

[FROM (PROG (TMP)

[SET FOR:LEFT (SETQ

RANGE:LEFT

(PLUS RANGE:LEFT (SETQ

TMP

(ITERUPDATE1])

(\* BY:TEST=NUMBER: THE "DYNAMIC"

CASE (SEE ITERDBY))

(AND (OR (NUMBERP BY:TEST)

(EQ BY:TEST (QUOTE DYNAMIC)))

)

(SETQ BY:TEST TMP])

(ITERROR ITERUPDATE))

(AND RANGE:OLD (SET RANGE:OLD RANGE:LEFT])

ITERUPDATE1

-----

[LAMBDA NIL (AND (EQ RANGE:OF (QUOTE IN))

(SET FOR:LEFT RANGE:LEFT))

(SEGEVAL BY:LEFT BY:RIGHT])

SEGEVAL

-----

[LAMBDA (\*\*L\* \*\*R\*)

(SELECTQ \*\*R\* ((CONSTANT NUMBER)

\*\*L\*)

(FUNCTION (APPLY\* \*\*L\* (EVAL FOR:LEFT)))

(SEGMAP \*\*L\* \*\*R\* (FUNCTION EVAL])

SEGMAP

-----

[LAMBDA (\*\*L\* \*\*R FN \*\*V)

(COND ((EQ \*\*L \*\*R)

\*\*V)

((NLISTP \*\*L)

(ITERROR SEGMAP))

(T (SEGMAP (CDR \*\*L)

\*\*R FN (APPLY\* FN (CAR \*\*L])

This function has been redefined, compare the discussion in 7.4.3.1.

The following list contains those variabls which are free in these functions and which are bound to values from the parsing step.

```
(FOR:LEFT FOR:RIGHT RANGE:OF RANGE:LEFT RANGE:RIGHT
RANGE:OLD BY:LEFT BY:RIGHT BY:TEST TO:LEFT TO:RIGHT
BIND:LEFT BIND:RIGHT MAIN:OF MAIN:LEFT MAIN:RIGHT
WHILE:OF WHILE:LEFT WHILE:RIGHT RPTWHILE:OF
RPTWHILE:LEFT RPTWHILE:RIGHT WHEN:OF WHEN:LEFT
WHEN:RIGHT FIRST:LEFT FIRST:RIGHT EACHTIME:LEFT
EACHTIME:RIGHT FINALLY:LEFT FINALLY:RIGHT ITER:BINDINGS)
```

We will here briefly describe how an iterative statement is executed. The numbers are found in the function definition of the iterxct-function.

The main function is iterxct.

- (1) The call to iterinit initializes the loop variables and the variable for the value returned from the statement, and may perform other initializations as well.
- (2) An expression in the iterative statement can be evaluated in every step in the loop before the exit-condition is tested.
- (3) In the selectg-expression the function iterxt checks the exit-condition and returns either T (finished), SKIP (skip this iteration) or other values (go on).
- (4) This expression executes the loop-body.
- (5) In this selectg-expression various operations may be performed depending on the format of the iterative statement.
- (6) A check can also be performed after the loop of the exit-condition.
- (7) The function iterupdate performs necessary updates of variables before the next iteration.
- (8) An expression can be evaluated before the exit from the iterative statement.
- (9) A return is made with the appropriate value.

These executor functions are all declared to be of the function class OPEN and open class BETA, i.e. they will all be beta-expanded.

Let us follow the example

```
(FOR I FROM 1 TO 10 DO (PRINT I))           (I.S 1)
```

In section 7.4.3.3 a problem was discussed where the analysis of assignments in the loop was not good enough. We will first show the example where we have excluded this variable by:test which caused the trouble. Later on in this appendix we will show why that problem occurred.

If the above example is given to the parser in the iterative statement implementation the 32 free variables will be bound and we can by them create the following prog-expression

```
[PROG [(FOR:LEFT (QUOTE I))
      [FOR:RIGHT (QUOTE (FROM 1 TO 10 DO (PRINT I)
      (RANGE:OP (QUOTE FROM))
      (RANGE:LEFT 1)
      (RANGE:RIGHT (QUOTE NUMBER))
      (RANGE:OLD NIL)
      (BY:LEFT 1)
      (BY:RIGHT (QUOTE NUMBER))
      (BY:TEST (QUOTE GREATERP))
      (TO:LEFT 10)
      (TO:RIGHT (QUOTE NUMBER))
      (BIND:LEFT NIL)
      (BIND:RIGHT NIL)
      (MAIN:OP (QUOTE DO))
      [MAIN:LEFT (QUOTE ((PRINT I)
      (MAIN:RIGHT NIL)
      (WHILE:OP NIL)
      (WHILE:LEFT NIL)
      (WHILE:RIGHT NIL)
      (RPTWHILE:OP NIL)
      (RPTWHILE:LEFT NIL)
      (RPTWHILE:RIGHT NIL)
      (WHEN:OP NIL)
      (WHEN:LEFT NIL)
      (WHEN:RIGHT NIL)
      (FIRST:LEFT NIL)
      (FIRST:RIGHT NIL)
      (EACHTIME:LEFT NIL)
      (EACHTIME:RIGHT NIL)
      (FINALLY:LEFT NIL)
      (FINALLY:RIGHT NIL)
      (ITER:BINDINGS (QUOTE (I)
      (RETURN (PROG (I)
                  (RETURN (ITERXCTI]
                  (RETURN (ITERXCTI]
```

This expression is given to redform.

We follow the beta-expansion of some functions

(ITERINIT)

gives

```
(QUOTED (PROGN (SETQ RANGE:LEFT 1)
               (SETQ I 1)
               (SETQ TO:LEFT 10))
  NIL :SIDE NIL NIL ((TO:LEFT :VALUE . 10)
    (I :VALUE . 1)
    (RANGE:LEFT :VALUE . 1)))
```

(ITERXT)

gives

```
(QUOTED (GREATERP RANGE:LEFT 10)
  (:VALUES T NIL)
  NIL NIL NIL)
```

(ITERUPDATE1)

(called from iterupdate)

gives

```
(QUOTED 1 (:VALUE . 1))
```

(ITERUPDATE)

gives

```
(QUOTED (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
  NIL :SIDE NIL NIL ((TMP :VALUE . 1)
    (RANGE:LEFT :DATATYPE . NUMBER)
    (I :DATATYPE . NUMBER)))
```

(ITERXCT)

gives them finally

```
(FROG (ITER:VALUE)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  (SETQ TO:LEFT 10)
  $$LP(SELECTQ (GREATERP RANGE:LEFT 10)
    (T (GO $$OUT))
    NIL)
  (PRINT I)
  (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
  (GO $$LP)
  $$OUT
  (RETURN ITER:VALUE))
```

The original expression (1) which was given to redform has now been reduced to

```

[PROG
  [(FOR:LEFT (QUOTE I))
   [FOR:RIGHT (QUOTE (FROM 1 TO 10 DO (PRINT I)
    (RANGE:OP (QUOTE FROM))
    (RANGE:LEFT 1)
    (RANGE:RIGHT (QUOTE NUMBER))
    (RANGE:OLD NIL)
    (BY:LEFT 1)
    (BY:RIGHT (QUOTE NUMBER))
    (BY:TEST (QUOTE GREATERP))
    (TO:LEFT 10)
    (TO:RIGHT (QUOTE NUMBER))
    (BIND:LEFT NIL)
    (BIND:RIGHT NIL)
    (MAIN:OP (QUOTE DO))
    [MAIN:LEFT (QUOTE ((PRINT I)
    (MAIN:RIGHT NIL)
    (WHILE:OP NIL)
    (WHILE:LEFT NIL)
    (WHILE:RIGHT NIL)
    (RPTWHILE:OP NIL)
    (RPTWHILE:LEFT NIL)
    (RPTWHILE:RIGHT NIL)
    (WHEN:OP NIL)
    (WHEN:LEFT NIL)
    (WHEN:RIGHT NIL)
    (FIRST:LEFT NIL)
    (FIRST:RIGHT NIL)
    (EACHTIME:LEFT NIL)
    (EACHTIME:RIGHT NIL)
    (FINALLY:LEFT NIL)
    (FINALLY:RIGHT NIL)
    (ITER:BINDINGS (QUOTE (I)
    (RETURN
      (PROG (I)
        (RETURN (PROG (ITER:VALUE)
          (SETQ RANGE:LEFT 1)
          (SETQ I 1)
          (SETQ TO:LEFT 10)
          $$LF
          (SELECTQ (GREATERP RANGE:LEFT
                    10)
                    (T (GO $$OUT))
                    NIL)
          (PRINT I)
          (SETQ I (SETQ RANGE:LEFT
                    (PLUS
                     RANGE:LEFT 1)))
          (GO $$LF)
          $$OUT
          (RETURN ITER:VALUE)]

```



These prog-expressions can now be collapsed into an expression by using the rule

$$(\text{PROG } (a_1 \dots a_n) (\text{RETURN } (\text{PROG } (b_1 \dots b_p) \text{stm}_1 \dots \text{stm}_k))) \rightarrow (\text{PROG } (a_1 \dots a_n \ b_1 \dots b_p) \text{stm}_1 \dots \text{stm}_k)$$

which can be applied as long as no variables in the inner prog are initialized by a form containing a variable bound in the outer prog.

A postprog-transformation will remove those prog-variables which will never be accessed and assignments to such variables. From the outermost prog only rang:left will remain we will receive

```
(PROG ((RANGE:LEFT 1)
      I ITER:VALUE)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  $$LP(SELECTQ (GREATERP RANGE:LEFT 10)
        (T (GO $$OUT))
        NIL)
  (PRINT I)
  (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
  (GO $$LP)
  $$OUT
  (RETURN ITER:VALUE))
```

Next postprog-transformation to apply is to remove initializations of variables which will be assigned again before the first label. In this case it seems better to leave the initialization of rang:left in the prog-variable list and to remove the assignment instead. In many other cases in this experiment a variable was initialized and then assigned to another value and therefore we have found it more suitable to remove the initializations.

```

( PROG (RANGE:LEFT I ITER:VALUE)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  $$LP (SELECTQ (GREATERP RANGE:LEFT 10)
    (T (GO $$OUT))
    NIL)
    (PRINT I)
    (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
    (GO $$LP)
  $$OUT
  (RETURN ITER:VALUE))

```

(4)

The system could not remove the iter:value variable because of the same problem which occurred with the by:test variable discussed later on in this appendix. Another run through REDFUN-2 will however produce

```

( PROG (RANGE:LEFT I)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  $$LP (SELECTQ (GREATERP RANGE:LEFT 10)
    (T (GO $$OUT))
    NIL)
    (PRINT I)
    (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
    (GO $$LP)
  $$OUT
  (RETURN NIL))

```

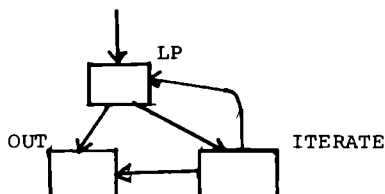
(5)

Some figures. A specialization of an iterative statement by REDFUN-2 takes about 20 seconds. REDFUN-2 has then been compiled straightforward by tcompl in INTERLISP. Some more efficiency can be obtained by block-compilation and compilation of all procedures stored on property lists, such as reducers, semantic procedures and collapsers. The executor functions in the iterative statement takes about 8 seconds to compile, as comparison. Figures from the reduced code follows in the table

	compiled code	interpreted code
iterative statement	0.21	0.68
iterative statement specialized by REDFUN-2	0.07	0.12
CLISP (as comparision)	0.07	0.40 (first translation) 0.12

(figures in sec. and obtained  
from runs on the DEC/20)

If we are not eliminating the variable by:test from the variable analysis, the following happens. The loop-structure in iterxt can be described by the directed graph



and we have a loop containing the blocks LP and ITERATE. The analysis of variables will find the following variables which may be assigned in that loop

iter:value, itertmp, i, range:left and by:test

but it will be impossible to extract any information about them.

This means that at the entry to the loop at label \$\$LP there will be no knowledge about the variable by:test and the reduction will result in

```
(PROG (RANGE:LEFT (BY:TEST (QUOTE GREATERP))
      I ITER:VALUE)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  $$LP(SELECTQ ISELECTQ BY:TEST
        ((NEVER DYNAMIC)
         NIL)
        (NLISTP (NLISTP RANGE:LEFT))
        (GREATERP (GREATERP RANGE:LEFT 10))
        (LESSP (LESSP RANGE:LEFT 10))
        (COND
          ((NOT (NUMBERP BY:TEST))
           (ITERATOR ITERXT))
          ((ZEROP BY:TEST)
           T)
          ((LESSP BY:TEST 0)
           (LESSP RANGE:LEFT 10))
          (T (GREATERP RANGE:LEFT 10)
           (T (GO $$OUT))
            (SKIP (GO $$ITERATE))
            NIL)
        (PRINT I)
  $$ITERATE
    (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
    (AND (OR (NUMBERP BY:TEST)
              (EQ BY:TEST (QUOTE DYNAMIC)))
         (SETQ BY:TEST 1))
    (GO $$LP)
  $$OUT
    (RETURN ITER:VALUE))
```

Another analysis and reduction can simplify the expression more. The analysis about by:test will now report that it can have either the value GREATERP (initialized in the prog-variable list) or 1 (from an assignment in the loop).

```
(PROG (RANGE:LEFT (BY:TEST (QUOTE GREATERP))
      I)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  $$LP(SELECTQ (SELECTQ BY:TEST
                    (GREATERP (GREATERP RANGE:LEFT 10))
                    (GREATERP RANGE:LEFT 10))
        (T (GO $$OUT))
        NIL)
    (PRINT I)
    (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 1)))
    (AND (NUMBERP BY:TEST)
         (SETQ BY:TEST 1))
    (GO $$LP)
  $$OUT
  (RETURN NIL))
```

No further reduction can be done by REDFUN-2 and we have reached the situation which was discussed and analysed in section 7.4.3.3.

Another possible simplification would follow from recognizing the common subexpression in the selectq, i.e. the two instances of (GREATERP RANGE:LEFT 10). However we have not attempted to include in REDFUN-2 any ability to recognize common subexpressions or to perform reductions based on them.

Further examples:

```
(FOR I FROM 1 TO N BY 2 SUM I)           (I.S 2)
```

will cause following code to be produced

```
(PROG (RANGE:LEFT TO:LEFT I ITEMP ITER:VALUE)
  (SETQ RANGE:LEFT 1)
  (SETQ I 1)
  (SETQ TO:LEFT N)
  (SETQ ITER:VALUE 0)
  $$LP
  (SELECTQ (GREATERP RANGE:LEFT TO:LEFT)
    (T (GO $$OUT))
    NIL)
  (SETQ ITEMP I)
  (SETQ ITER:VALUE (PLUS ITER:VALUE ITEMP))
  (SETQ I (SETQ RANGE:LEFT (PLUS RANGE:LEFT 2)))
  (GO $$LP)
  $$OUT
  (RETURN ITER:VALUE))
```

Remark. The itertmp-variable could be propagated, i.e. replacing the occurrences of the variable to the form which binds it, in this case the form i. The variable can then be deleted from the prog-variable list. This kind of propagation can be performed by a postprog-transformation but has not been included yet in our system.

```
(NEVER (ATOM X)
      FOR X IN LLIST) (I.S 3)
```

```
(PROG (RANGE:LEFT X ITERTMP ITER:VALUE)
      (SETQ RANGE:LEFT LLIST)
      (SETQ X (CAR RANGE:LEFT))
      (SETQ ITER:VALUE T)
      $$LP
      (SELECTQ (NLISTP RANGE:LEFT)
                (T (GO $$OUT))
                NIL)
      (SETQ ITERTMP (ATOM X))
      (COND (ITERTMP (SETQ ITER:VALUE NIL)
                (GO $$OUT)))
      [SETQ X (CAR (SETQ RANGE:LEFT
                    (PROGN (SETQ X RANGE:LEFT)
                          (CDR X))
                    (GO $$LP))
      $$OUT
      (RETURN ITER:VALUE))
```

Remark. This statement is true if LLIST does not contain any atoms. A propagation of the variable x inside the progn-expression would clean up among the assignments of the loop variables x and range:left.

(IN AL COLLECT CAR)

(I.S 4)

```

(PROG (RANGE:LEFT DUMMYIV ITER:VALUE)
  (SETQ RANGE:LEFT AL)
  (SETQ DUMMYIV (CAR RANGE:LEFT))
  $$LP
  (SELECTQ (NLISTP RANGE:LEFT)
    (T (GO $$OUT))
    NIL)
  (SETQ ITER:VALUE (CAR DUMMYIV))
  (SETQ ITER:VALUE (CONS ITER:VALUE))
  (SETQ DUMMYIV (CAR (SETQ RANGE:LEFT
    (PROGN (SETQ DUMMYIV
      RANGE:LEFT)
      (CDR DUMMYIV)
    (GO $$LP)
    $$OUT
    (RETURN (DREVERSE ITER:VALUE)))

```

Remark. Performs

```
(MAPCAR AL (FUNCTION CAR))
```

A dummy-variable dummyiv is introduced and defined to the nalaysis program to be a variable which varies in the loop.



```

(FOR X IN L BIND Y FIRST (SETQ Y 0)      (I.S 5)
  DO
    [COND ((ATOM Y)
            (SETQ Y (ADD1 Y)
          FINALLY
            (RETURN Y))

(PROG (RANGE:LEFT X Y)
  (SETQ RANGE:LEFT L)
  (SETQ X (CAR RANGE:LEFT))
  (SETQ Y 0)
  $$LP
  (SELECTQ (NLISTP RANGE:LEFT)
    (T (GO $$OUT))
    NIL)
  [COND ((ATOM X)
          (SETQ Y (ADD1 Y)
        [SETQ X (CAR (SETQ RANGE:LEFT
                      (PROGN (SETQ X RANGE:LEFT)
                            (CDR X))
      (GO $$LP)
  $$OUT
  (RETURN Y))

```

Remark. The statement counts the number of atoms in a list. This example is evaluated incorrectly by the iterative statement routines. This depends probably of the return-statement after FINALLY which causes a return out of the wrong prog. Interesting, however, is that this reduced version is correct!

```

[FOR X IN L JOIN (CDR X)                                (I.S 6)
  WHEN
    (AND (LISTP X)
      (EQ (CAR X)
        (QUOTE A)

```

```

(PROG
  (RANGE:LEFT X ITERTMP ITER:VALUE)
  (SETQ RANGE:LEFT L)
  (SETQ X (CAR RANGE:LEFT))
  $$LP
  (SELECTQ
    (OR (NLISTP RANGE:LEFT)
      (AND (NEQ T (AND (LISTP X)
        (EQ (CAR X)
          (QUOTE A))
        T))
      (QUOTE SKIP)))
    (T (GO $$OUT))
    (SKIP (GO $$ITERATE))
    NIL)
  (SETQ ITERTMP (CDR X))
  [COND [(NLISTP ITER:VALUE)
    (SETQ ITER:VALUE (CONS ITERTMP (LAST ITERTMP)
      (T (RPLACD (CDR ITER:VALUE)
        ITERTMP)
        (RPLACD ITER:VALUE (LAST (CDR ITER:VALUE)
  $$ITERATE
  [SETQ X (CAR (SETQ RANGE:LEFT (PROGN (SETQ X
    RANGE:LEFT)
    (CDR X)

  (GO $$LP)
  $$OUT
  (RETURN (CAR ITER:VALUE)))

```

Remark. This statements concatenates those sublists in the list L, in which the first element is the atom A.

## APPENDIX V

## PROGRAM CODE FROM SOME CENTRAL FUNCTIONS IN REDFUN-2

The program code for the central functions in REDFUN-2, redform, redargs and redfun and the function tryapply are given here. The code is also given for the and-reducer and its auxiliary functions and a table shows the various transformations that reducer performs.

```

(DEFINEQ
  (REDFORM
    [LAMBDA (FORM AL RCTXT)

      (* * Extracts the reduced expression
      from the QUOTED-expression)

      (UNQUOTED (REDFORMX FORM AL RCTXT))

  (REDFORMX
    [LAMBDA (FORM AL RCTXT)

      (* * FORM is reduced with a-list AL in
      context RCTXT)

      (PROG (TEMP)
        (OR RCTXT (SETQ RCTXT (QUOTE V)))
        (RETURN
          (COND
            ((NULL FORM)
              NIL)
            ((LITATOM FORM)
              (COND
                ((EQ FORM T)
                  T)
                ((SETQ TEMP (SASSOCBIN FORM AL))
                  (PROG (TMP)
                    (* FORM is a
                    variable with
                    VALUE-DESCRIPTOR
                    (excl NOBIN) on
                    a-list)

                    (RETURN
                      (COND
                        ((SUBSTVALUE TEMP)
                          (* A SUBSTITUTION
                          DESCRIPTOR)
                          (SUBSTVAR FORM TEMP AL
                            RCTXT))
                        ((SETQ TMP
                          (KNOWNVALUE (CDR TEMP)))
                          (* Known value of
                          variable (a
                          :VALUE-descriptor))
                          (QWOTE (CDR TMP)))
                        (T
                          (* Check for FALSE
                          and TRUE branch
                          contexts)
                          (MAKEQUOTED
                            FORM
                            (CDR TEMP)
                            NIL NIL
                            (COND
                              ((TRUECTXTF RCTXT)
                                (MAKEVARTRUEVAL
                                  FORM)))
                            ))
                          ))
                    ))
                ))
            ))
          ))
      )
    )
  )

```

```

(COND
  ((FALSECTXTP RCTXT)
    (MAKEVARFALSEVAL
      FORM)
  ((TRUEORFALSECTXTP RCTXT)
    (* Check for FALSE
      and TRUE branch
      contexts)
    (MAKEQUOTED FORM NIL NIL NIL
      (COND
        ((TRUECTXTP RCTXT)
          (MAKEVARTRUEVAL FORM)))
        (COND
          ((FALSECTXTP RCTXT)
            (MAKEVARFALSEVAL FORM)
          (T FORM))))
  ((NLISTP FORM)
    FORM)
  ((QUOTE P FORM)
    (* FORM is a
      QUOTE-expression)
    (APPLY* (GETP (QUOTE QUOTE)
      (QUOTE REDUCER))
      FORM))
  ((QUOTEDP FORM)
    (* FORM is a
      QUOTED-expression.
      Leave it)
    (COND
      ((ONLYNILS (CDDR FORM))
        (UNQUOTED FORM))
      (T FORM)))
  ((NLISTP (CAR FORM))
    ECOND
    ((MEMB (CAR FORM)
      FORCEREDARGSLIST)
      (* Reduce arguments
        before
        classification of
        the FORM)
      (SETQ FORM
        (CONS
          (CAR FORM)
          (QUOTEDALL
            (REDARGSTX (CDR FORM)
              AL
              (FNCTXT (CAR FORM)
                RCTXT)
            (* * Branch on function-class.
              Collaps the result)

          (COLLAPS
            (SELECTQ
              (CLASSIFY FORM)
              (PURE (TRYAPPLY (CAR FORM)
                (REDARGSX (CDR FORM)
                  AL
                  (QUOTE V))

```

```

                                RCTXT))
(OPEN
  (EXPAND (CAR FORM)
    (QUOTEDALL
      (REDARGSX (CDR FORM)
        AL
        (QUOTE V)))
    AL RCTXT))
  (REDUCER (COND
    ((SETQ TEMP
      (GETP (CAR FORM)
        (QUOTE REDUCER)))
      (APPLY* TEMP FORM AL RCTXT)
    )
    (T FORM)))
  (EXPR (EXPRFN (CAR FORM)
    (REDARGSX (CDR FORM)
      AL RCTXT)))
  (SIDEEXPR
    (SIDEEXPRFN
      (CAR FORM)
      (REDARGSCTX (CDR FORM)
        AL
        (FNCTXT (CAR FORM)
          RCTXT))
      RCTXT))
    (SIDEFEXPR (SIDEFEXPRFN FORM))
    FORM)
  AL RCTXT))
(T
  (* CAR of FORM is
  non-atom)
  (SELECTQ
    (CAR (SETQ TEMP (REDFUNX (CAR FORM)
      AL RCTXT)))
    [NLAMBDA
      (* A check for
      assignments ought to
      be done)
      (CONS TEMP (CDR FORM))
      (OPENLAMBDA
        (EXPAND TEMP
          (QUOTEDALL (REDARGSX
            (CDR FORM)
            AL
            (QUOTE V)))
          AL RCTXT))
      (TRYAPPLYLAMBDA TEMP (REDARGSX
        (CDR FORM)
        AL
        (QUOTE V))
        RCTXT)]
  (REDARGS
    [LAMBDA (ARGS AL RCTXT)

      (* * Extracts the reduced expression
      from the QUOTED-expression)

```

```

(MAPUNQUOTED (REDARGSX ARGS AL RCTXT))

(REDARGSX
  [LAMBDA (ARGS AL RCTXT)
    (* Performs REDFORMX
    on each element on
    ARGS)

    (PROG (RES TEMP)
      (COND
        ((NULL ARGS)
          (RETURN NIL)))
      LOP (SETQ TEMP (REDFORMX (CAR ARGS)
                              AL RCTXT))
        (SETQ RES (CONS TEMP RES))
        (SETQ ARGS (CDR ARGS))
      [COND
        ((NULL ARGS)
          (RETURN (DREVERSE RES)
            (* Transfer
            assigninfo from
            previous FORM)
            (SETQ AL (ADDSSETQAL (GETSETQ TEMP)
                                AL)))
        (GO LOP)])

    (GO LOP)])

(REDFUN
  [LAMBDA (EXPR AL RCTXT)
    (* * Extracts the reduced expression
    from the QUOTED-expression)

    (UNQUOTED (REDFUNX EXPR AL RCTXT))]

(REDFUNX
  [LAMBDA (EXPR AL RCTXT)
    (* * Reduces a function expression)

    (COND
      ((NLISTP EXPR)
        EXPR)
      (T
        (SELECTQ
          (CAR EXPR)
          [[LAMBDA NLAMBDA]
            (SETQ AL (REMAI (CADR EXPR)
                            AL))
            (PROG (TEMP)
              (SETQ TEMP (REDARGSX (CDR EXPR)
                                    AL RCTXT))
              (RETURN
                (MARKSIDED (MARKSETQ
                            (LINS (CAR EXPR)
                                (CADR EXPR)
                                (MAPUNQUOTED TEMP))
                            (CHECKSETQARGS TEMP T))
                (MAPSIDED TEMP)]
          ]))

```

```

(FUNARG (REDFUNX (CADR EXPR)
  (ADDDVALUE (MKALIST (CDDR EXPR))
    AL)
  RCTXT))
EXPR])

(TRYAPPLY
  CLAMBDA (FN ARGS RCTXT)
    (PROG (VAL CLASS RES)
      [COND
        ((NOVALUECTXTF RCTXT)          (* NOVALUE-context)
          (RETURN (EXTRACTSIDEDARGS ARGS NIL
            (QUOTE NIL)
          )
        )
      ]
      [COND
        ((NULL ARGS)
          (RETURN (QWOTE (APPLY FN)
            (* RES is used to bound the resulting
              expression. If a single value exists
              then an immediate return is made.
              Otherwise a QUOTED-expression is made or
              NIL , if NIL other semantic procedures
              are applied)
          )
        )
      ]
    )
  )
  (SELECTQ
    (SETQ CLASS (CLASSIFYARGS ARGS))
    CALLQWOTED
      (* All arguments
        known and no
        side-effects)
      (RETURN (QWOTE (EVAL (CONS FN (MAPUNQUOTED
        ARGS)
      )
    )
  )
  (ALLSINGLEVALUE
    (* All arguments
      known but
      side-effects are
      involved)
    (RETURN
      (EXTRACTSIDEDARGS
        ARGS
        [QWOTE (EVAL (CONS FN (MAKEPUREARGS
          ARGS)
        )
      )
      RCTXT)))
    CALLKNOWNVALUES
      (* All arguments
        have known values
        (either :VALUE or
        :VALUES))
      (COND
        ((QWOTED
          (SETQ RES
            (MAKEEXPRESSION
              (CONS FN (MAPUNQUOTED ARGS))
              [PROG (PROC)
                (RETURN
                  (COND
                    ((SETQ PROC
                      (GETF FN
                        (QUOTE VALUEFN))
                    )
                  )
                )
              )
            )
          )
        )
      )
    )
  )

```



```

                                (APPLY PROC ARGS))
                                (T (EVALVALUES
                                    FN
                                    (MAPVALUEARGS ARGS)
                                    RCTXT)
                                (MAPSIDED ARGS)
                                NIL NIL NIL ARGS)))
                                (* A single value
                                and no side-effects)
                                (RETURN RES)
(ALLNOVALUES                                (* All arguments are
                                            of :NOVALUES-type)
  (SETQ RES
    (SELECTQ (CLASSIFYNOVALUES FN)
      (EVALNOVALUES
        (MAKEEXPRESSION
          (CONS FN (MAPUNQUOTED ARGS))
          (EVALNOVALUES FN (
            MAPNOVALUEARGS
              ARGS))
          (MAPSIDED ARGS)
          NIL NIL NIL ARGS))
        NIL)))
  NIL)
COR
RES                                (* NIL try semantic
                                Procedures)
(COND
  (COR (AND (MEMB CLASS (QUOTE (NOVALUES
                                ALLNOVALUES))))
    (SETQ VAL
      (APPLY (GETF FN (QUOTE NOVALUEFN))
        ARGS)))
    (AND (EQ CLASS (QUOTE DATATYPES))
      (SETQ VAL
        (APPLY (GETF FN (QUOTE
          DATATYPEPFN))
          ARGS)))
      (SETQ VAL (APPLY (GETF FN (QUOTE
        TRYAPPLYFN))
        (CONS RCTXT ARGS))
      (COND
        ((QUOTED (SETQ RES
          (MAKEEXPRESSION
            (CONS FN (MAPUNQUOTED ARGS))
            VAL
            (MAPSIDED ARGS)
            NIL NIL NIL ARGS)))
          (* A single value
          and no side-effects,
          so return)
          (RETURN RES)
        (AND
          (TRUEORFALSECTXTF RCTXT)
          (SETQ VAL (APPLY (GETF FN (QUOTE CTXTFN))
            (CONS RCTXT ARGS)))
          (* TRUECTXT- and/or
          FALSECTXT-

```

```

information is
available)
(* Transfer
side-effect and
assignment
information)

(RETURN
  (PROG ((SETQLIST (CTXTSETQARGS ARGS)))
    (RETURN (MAKEEXPRESSION
      (CONS FN (MAPUNQUOTED ARGS))
      (AND RES (GETVALUES RES))
      (MAPSIDED ARGS)
      (COLLECTTRUECTXT
        (CADR VAL)
        SETQLIST RCTXT)
      (COLLECTFALSECTXT
        (CADDR VAL)
        SETQLIST RCTXT)
      (CHECKSETQARGS ARGS T)
    OR RES (SETQ RES (COND
      ((SETQ VAL (OR (VALUEDATATYPE FN ARGS)
        (VALUESKNOWN FN)))
        (* Value or datatype
        information of the
        result is available)
      (MAKEEXPRESSION (CONS FN (MAPUNQUOTED
        ARGS))
        VAL
        (MAPSIDED ARGS)
        NIL NIL NIL ARGS))
      (T (CONS FN (MAPUNQUOTED ARGS)
    (RETURN (MARKSIDED (MARKSETQ RES
      (CHECKSETQARGS
        ARGS T))
      (MAPSIDED ARGS))

(ANDREDUCERAUX
  (LAMBDA (ARGS AL RCTXT)

    (* * Reduces arguments to
    AND-expressions.)

    (PROG ((CTXT RCTXT)
      (ALFACTXT (QUOTE ALFA1))
      FALSECTXTVARS LIST REDARG (RES (CONS)))

    (* CTXT and ALFACTXT used in order to
    change context. FALSECTXTVARS is a list
    of those variables for which
    falsectxt-info is possible and controls
    context changes.
    RES holds the new reduced argument
    list.)

    (COND
      ((NULL ARGS)

```

```

      (RETURN NIL)))
LOP [COND
  ((NULL (CDR ARGS))
    (RETURN
      (CAR (TCONC RES (REDFORMX
        (CAR ARGS)
        AL
        (ANDCTXT (QUOTE BETA)
          RCTXT ALFACTXT])
      [SETQ REDARG (REDFORMX (CAR ARGS)
        AL
        (SETQ CTXT
          (ANDCTXT ALFACTXT CTXT])
      [COND
        [(KNOWNFALSE REDARG)
          (* A FALSE ARG is
            found. Skip rest of
            ARGS)
          (RETURN (CAR (TCONC RES REDARG])
        [(KNOWNTTRUE REDARG)
          (* A TRUE ARG is
            found. If no
            side-effects remove
            it)
          (COND
            ((SIDED REDARG)
              (TCONC RES REDARG])
            (T (TCONC RES REDARG)
              (AND (FALSECTXT RCTXT)
                (FIX-FALSECTXT-IN-AND])
              (SETQ AL (ADIVALTRUECTXT REDARG AL))
              (SETQ ARGS (CDR ARGS))
              (GO LOP])
          (ANDIFY
            (LAMBDA (L RCTXT)

              (* L is reduced args in an
                AND-expression A new AND-expression is
                built up and stored in a
                QUOTED-expression)

              (COND
                ((NULL L)
                  T)
                ((NULL (CDR L))
                  (CAR L))
                ((KNOWNTTRUE (CAR L))
                  (* If first ARG is
                    TRUE, break it into
                    a PROGN)
                  (PROGNIFY (LIST (CAR L)
                    (ANDIFY (CDR L)
                      RCTXT))
                    RCTXT))
                (T
                  (PROG ((RES (TCONC NIL (QUOTE AND)))
                    LASTSIDEPOS TRUECTXTLIST FALSECTXTLIST
                    SETQLIST (FIRSTFORM T))

```

(\* The resulted AND-expression is built up in res. Lastsidepos points out last ARG performing a side-effect used if the whole expressions always will be FALSE.)

```

(* Try to calculate
FALSETXT-element if
in FALSE branch
context)
(AND (FALSETXTP RCTXT)
  (SETQ FALSETXTLIST (COLLANDFALSETXT
    L)))
LOP (* NCONC next
      argument to the
      AND-expression)
(TLCONC RES (QUOTE AND)
  (UNQUOTED (CAR L)))
(* Check if
side-effect occurs)
(AND (SIDED (CAR L))
  (SETQ LASTSIDEPOS (CDR RES)))
(* Save information
for the
TRUETXT-element)
(AND (TRUETXTP RCTXT)
  (CHECKTRUETXT)) (* Save information
about assignments)
(CHECKSETQ (CAR L)
  (COND
    (FIRSTFORM (SETQ FIRSTFORM NIL)
      T)))
ECOND
  ((NULL (CDR L)) (* Create the new
    QUOTED-expression)
    (RETURN
      (MAKEQUOTED
        (COND
          ((NOVALUECTXP RCTXT)
            (* In
            NOVALUE-context.
            Further reductions
            can be performed)
          (COND
            [LASTSIDEPOS
              (RPLACD LASTSIDEPOS NIL)
              (COND
                ((CDR RES)
                  (CAR RES))
                (T (CADR RES)
                  (T NIL)))
              (KNOWNFALSE (CAR L))
                (* Value of
                AND-expression is
                FALSE. More
                reductions can be
                performed)
              (COND
                ((NULL LASTSIDEPOS)

```

```

(CAR L))
(T (RPLACD LASTSIDEPOS
    (LIST NIL))
(COND
  ((CDAR RES)
   (CAR RES))
  (T (CDAR RES)
      (T (CAR RES)))
(CAND (VALUECTXTP RCTXT)
  (VALUED (CAR L))
  (MAKEFALSEALSO
   (GETVALUES (CAR L)
NIL
  (AND LASTSIDEPOS (QUOTE :SIDE))
  (AND (TRUECTXTP RCTXT)
        (NOT (KNOWNFALSE (CAR L)))
        (CTXTLISTRED TRUECTXTLIST))
  (AND (FALSECTXTP RCTXT)
        (CTXTLISTRED FALSECTXTLIST))
  SETQLIST]
  (SETQ L (CDR L))
  (GO LOOP]
)

(PUTPROPS AND REDUCER [LAMBDA (FORM AL RCTXT*)
  (* RCTXT* CAN BE RESET BY
   ANDCTXT)
  (ANDIFY (ANDREDUCERAUX
            (CDR FORM)
            AL RCTXT*)
            RCTXT*])

```

Transformations performed by the and-reducer

(AND)  $\rightarrow T$   
 (AND a)  $\rightarrow a$   
 (AND  $a_1 \dots a_k \dots a_n$ )  $\rightarrow$  (AND  $a_1 \dots a_{k-1} a_{k+1} \dots a_n$ )  
     if  $a_k$  is known to be true  
     and performs no side-effects  
 (AND  $a_1 \dots a_k \dots a_n$ )  $\rightarrow$  (AND  $a_1 \dots a_k$ )  
     if  $a_k$  is known to be false  
 (AND  $a_1 a_2 \dots a_n$ )  $\rightarrow$  (PROGN  $a_1$  (AND  $a_2 \dots a_n$ ))  
     if  $a_1$  is known to be true  
     and performs a side-effect  
 (AND  $a_1 \dots a_k \dots a_n$ )  $\rightarrow$  (AND  $a_1 \dots a_k$  NIL)  
     if  $a_n$  is known to be false and  
      $a_{k+1}$  to  $a_n$  performs no side-effects  
 (AND  $a_1 \dots a_{k-1}$  (AND  $b_1 \dots b_m$ )  $a_k \dots a_n$ )  $\rightarrow$   
      $\rightarrow$  (AND  $a_1 \dots a_{k-1} b_1 \dots b_m a_k \dots a_n$ )

In a novalue-context

(AND  $a_1 \dots a_k \dots a_n$ )  $\rightarrow$  (AND  $a_1 \dots a_k$ )  
     if  $a_n$  is known to be false and  
      $a_{k+1}$  to  $a_n$  perform no side-effects

## APPENDIX VI

## EXAMPLES FROM CHAPTER 6 RUN THROUGH THE REDFUN-2 PROGRAM

In this appendix we will show som examples from chapter 6 run through the REDFUN-2 program. These examples are marked as (EX 4) in that chapter and corresponds to EX 4 among the print outs.

```
*****
EX 1      (SECTION 6.3)
*****
```

```
FORM=
[AND (EQ X Y)
      (COND ((NUMBERP X)
              (FOO X))
             (T (SETQ Z (FIE X))
                 (SETQ V (QUOTE B))
```

```
AL=
  (Y . (:VALUE . A))
```

```
CONTEXT=VTF
```

```
REDUCED FORM=
[AND (EQ X (QUOTE A))
      (PROGN (SETQ Z (FIE (QUOTE A)))
              (SETQ V (QUOTE B))
```

```
VALUES=
  (:VALUES . (NIL B))
```

```
SIDE-EFFECT=YES
```

```
TRUECTXT=
  (V . (:SETQVALUE . (:VALUE . B)))
  (Z . NOBIN)
  (X . (:VALUE . A))
```

```
FALSECTXT=
  (X . (:NOVALUES . (A)))
```

```
ASSIGNINFO=
  (V . (:ADDVALUE . (:VALUE . B)))
  (Z . (:ADDVALUE . NOBIN))
```

```
*****
EX 2      (SECTION 6.4.3)
*****
```

```
FORM=
(EQ X Y)
```

```
AL=
  (X . (:VALUES . (A B)))
  (Y . (:VALUES . (C D)))
```

```
CONTEXT=V
```

```
REDUCED FORM=
NIL
```



```
*****
EX 3      (SECTION 6.4.3)
*****
```

```
FORM=
(EQ X Y)
```

```
AL=
  (X . (:VALUES . (A B)))
  (Y . (:VALUES . (A C)))
```

```
CONTEXT=V
```

```
REDUCED FORM=
(EQ X Y)
```

```
*****
EX 4      (SECTION 6.4.3)
*****
```

```
FORM=
(EQ X Y)
```

```
AL=
  (X . (:VALUES . (B C)))
  (Y . (:NOVALUES . (A B C)))
```

```
CONTEXT=V
```

```
REDUCED FORM=
NIL
```

```
*****
EX 5      (SECTION 6.4.3)
*****
```

```
FORM=
(MEMB X (QUOTE (A D)))
```

```
AL=
  (X . (:NOVALUES . (A B C D)))
```

```
CONTEXT=V
```

```
REDUCED FORM=
NIL
```

```
*****
EX 6      (SECTION 6.4.3)
*****
```

```
FORM=
(EQ X (QUOTE A))

AL=
  (X , (:DATATYPE , INTEGER))

CONTEXT=V
```

```
REDUCED FORM=
NIL
```

```
*****
EX 7      (SECTION 6.5.3)
*****
```

```
FORM=
(COND ((EQ X (QUOTE A))
      (F1 X))
      ((MEMB X (QUOTE (A B)))
      (F2 X))
      ((EQ X (QUOTE B))
      (F3 X))
      (T (F4 X)))
```

```
AL=
  (X , (:VALUES , (A B C)))

CONTEXT=V
```

```
REDUCED FORM=
[COND ((EQ X (QUOTE A))
      (F1 (QUOTE A)))
      ((MEMB X (QUOTE (A B)))
      (F2 (QUOTE B)))
      (T (F4 (QUOTE C))
```

```
*****
EX 8      (SECTION 6.5.5.2)
*****
```

```
FORM=
(COND ((EQ X 5)
      T)
      ((EQ X 7)
      NIL)
      (T Y))
```

```
AL=
```

```
CONTEXT=VTF
```

```
REDUCED FORM=
(COND ((EQ X 5)
      T)
      ((EQ X 7)
      NIL)
      (T Y))
```

```
SIDE-EFFECT=NO
```

```
TRUEXT=
(X . (:NOVALUES . (7)))
```

```
FALSEXT=
(X . (:NOVALUES . (5)))
```

```
*****
EX 9      (SECTION 6.5.5.3)
*****
```

```
FORM=
(COND ((EQ X 3)
        NIL)
      ((EQ X 5)
        (EQ Y 5))
      ((EQ Y 3)
        T)
      (T NIL))
```

```
AL=
```

```
CONTEXT=VTF
```

```
REDUCED FORM=
(COND ((EQ X 3)
        NIL)
      ((EQ X 5)
        (EQ Y 5))
      ((EQ Y 3)
        T)
      (T NIL))
```

```
SIDE-EFFECT=NO
```

```
TRUEXT=
(X . (:NOVALUES . (3)))
(Y . (:VALUES . (5 3)))
```

```
*****
EX 10      (SECTION 6.6)
*****
```

```
FORM=
(MEMB (SETQ X (QUOTE A))
      (CONS (SETQ Y (QUOTE C))
             Z))
```

```
AL=
  (Z . (:VALUE . (B A)))
```

```
CONTEXT=V
```

```
REDUCED FORM=
(PROGN (SETQ X (QUOTE A))
       (SETQ Y (QUOTE C))
       (QUOTE (A)))
```

```
VALUES=
  (:VALUE . (A))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
  (Y . (:VALUE . C))
  (X . (:VALUE . A))
```

```
*****
EX 11      (SECTION 6.7.2)
*****
```

```
FORM=
(COND ((EQ X Y)
      (SETQ Y 5)
      (FOO X Y))
      (T (FIE Y)))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(COND ((EQ X Y)
      (SETQ Y 5)
      (FOO X 5))
      (T (FIE Y)))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
  (Y . (:ADDDVALUE . (:VALUE . 5)))
```

```
*****
EX 12      (SECTION 6.7.2)
*****
```

```
FORM=
(COND ((NULL X)
      (FOO Y))
      ((EQ X (SETQ Y 5))
       (FIE X Y))
      (T (FUM Y)))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(COND ((NULL X)
      (FOO Y))
      ((EQ X (SETQ Y 5))
       (FIE 5 5))
      (T (FUM 5)))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(Y . (:ADDDVALUE . (:VALUE . 5)))
```

```
*****
EX 13      (SECTION 6.7.2)
*****
```

```
FORM=
(COND ((EQ X (SETQ Z 5))
      (FOO X Z))
      (T (FIE X Z)))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(COND ((EQ X (SETQ Z 5))
      (FOO 5 5))
      (T (FIE X 5)))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(Z . (:VALUE . 5))
```

```
*****
EX 14  (SECTION 6.7.2)
*****
```

```
FORM=
(AND (EQ X (SETQ Y 5))
      (FOO X Z))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(AND (EQ X (SETQ Y 5))
      (FOO 5 Z))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(Y , (:VALUE , 5))
```

```
*****
EX 15  (SECTION 6.7.2)
*****
```

```
FORM=
(AND (OR L (SETQ Y 5))
      (FOO Y))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(AND (OR L (SETQ Y 5))
      (FOO Y))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(Y , (:ADDDVALUE , (:VALUE , 5)))
```

```
*****
EX 16      (SECTION 6.7.2)
*****
```

```
FORM=
(COND ((NULL X)
      (SETQ Y 1))
      ((EQ X Y)
      (SETQ Y 2))
      (T (SETQ Y 3)))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(COND ((NULL X)
      (SETQ Y 1))
      ((EQ X Y)
      (SETQ Y 2))
      (T (SETQ Y 3)))
```

```
VALUES=
(:VALUES . (3 2 1))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(Y . (:VALUES . (1 2 3)))
```



```
*****
EX 17      (SECTION 6.7.2)
*****
```

FORM=

```
(COND ((NULL X)
      (SETQ Y 1))
      ((EQ X Y)
      (SETQ Y 2))
      ((FOO (SETQ Y 3))
      (FIE X Y))
      (T (FUM X Y)))
```

AL=

CONTEXT=V

REDUCED FORM=

```
(COND ((NULL X)
      (SETQ Y 1))
      ((EQ X Y)
      (SETQ Y 2))
      ((FOO (SETQ Y 3))
      (FIE X 3))
      (T (FUM X 3)))
```

SIDE-EFFECT=YES

ASSIGNINFO=

```
(Y . (:VALUES . (1 2 3)))
```

```
*****
EX 18      (SECTION 6.7.2.1)
*****
```

```
FORM=
(COND ((SETQ X NIL))
      ((PUT A B NIL))
      ((FOO X)
       (SETQ Y T))
      ((SETQ Y NIL))
      (T (FUM X Y)))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
[PROGN (SETQ X NIL)
      (PUT A B NIL)
      (COND ((FOO NIL)
              (SETQ Y T))
            ((SETQ Y NIL))
            (T (FUM NIL NIL)))]
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(Y . (:VALUES . (T NIL)))
(X . (:VALUE . NIL))
```

```
*****
EX 19      (SECTION 6.7.2.2)
*****
```

```
FORM=
(SETQ X 5)
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(SETQ X 5)
```

```
VALUES=
(:VALUE . 5)
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(X . (:VALUE . 5))
```

```
*****
EX 20      (SECTION 6.7.2.2)
*****
```

```
FORM=
(FOO (CONS (SETQ X A)
           B)
      (SETQ Y NIL))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(FOO (CONS (SETQ X A)
           B)
      (SETQ Y NIL))
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(Y . (:VALUE . NIL))
(X . NOBIN)
```

```
*****
EX 21      (SECTION 6.7.4)
*****
```

```
FORM=
(SET (SETQ VAR (QUOTE X))
     10)
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(PROGN (SETQ VAR (QUOTE X))
       (SETQ X 10))
```

```
VALUES=
(:VALUE . 10)
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(X . (:VALUE . 10))
(VAR . (:VALUE . X))
```

```
*****
EX 22      (SECTION 6.7.4)
*****
```

```
FORM=
(PROGN [COND ((FOO X)
              (SETQ VAR (QUOTE A)))
        ((FIE X)
         (SETQ VAR (QUOTE B)))
        (T (SETQ VAR (QUOTE C))
          (SET VAR 10))
```

```
AL=
```

```
CONTEXT=V
```

```
REDUCED FORM=
(PROGN [COND ((FOO X)
              (SETQ VAR (QUOTE A)))
        ((FIE X)
         (SETQ VAR (QUOTE B)))
        (T (SETQ VAR (QUOTE C))
          (SET VAR 10))
```

```
VALUES=
(:VALUE . 10)
```

```
SIDE-EFFECT=YES
```

```
ASSIGNINFO=
(A . (:ADDDVALUE . (:VALUE . 10)))
(B . (:ADDDVALUE . (:VALUE . 10)))
(C . (:ADDDVALUE . (:VALUE . 10)))
(VAR . (:VALUES . (A B C)))
```

## REFERENCES

- ALL72 Allen, F. and Cocke, J. A catalogue of optimizing transformations, R. Rustin (Ed.) Design and Optimization of Compilers, Prentice-Hall, 1972
- BEC76 Beckman, L., Haraldson, A., Oskarsson, Ö. and Sandewall, E. A Partial Evaluator, and its Use as a Programming Tool, Artificial Intelligence Journal 7, (1976) 319-357
- BUR75 Burstall, R.M. and Darlington, J. Some transformations for developing recursive programs, Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, 1976
- CHE72 Cheatham, T.E. and Wegbreit, B. A laboratory for the study of automatic programming, AFIPS Conf. Proc., Vol 70, 1972
- DAR72 Darlington, J. A semantic approach to automatic program improvement, Experimental Programming Reports: No 27, School of Artificial Intelligence, University of Edinburgh, 1972
- DAR73 Darlington, J. and Burstall, R.M. A system which automatically improves programs, Proceedings of Third International Joint Conference on Artificial Intelligence Stanford, Calif, 1973
- DIJ70 Dijkstra, E.W. Structured Programming, Software Engineering techniques, J.N. Buxton and B. Randell (Eds.) NATO Scientific Affairs Division, Brussels, Belgium, 1970

- GRI71 Gries, D. Compiler Construction for Digital Computers, John Wiley & Sons Inc., 1971
- HAG76 Häggglund, S. and Oskarsson, Ö. En teknik för utformning av användardialoger i interaktiva datasystem (in Swedish), Informatics Laboratory, Linköpings University, Lith-MAT-R-76-13, 1976
- HAR73 Haraldson, A. and Sandewall, E. PCDB System Documentation Part I, Datalogilaboratoriet, Uppsala University, Sweden, DLU 73/9, 1973
- HAR74 Haraldson, A. PCDB - A procedure generator for a predicate calculus data base, Information Processing 74, North-Holland Publishing Company, 1974
- INT75 INTERLISP/360 AND /370 USER REFERENCE MANUAL, Uppsala University Data Center, Uppsala, Sweden, 1975
- KNU74 Knuth, D. Structured Programming with goto Statements, ACM Computing Surveys 6, 4, 1974
- LOV76 Loveman, D.B. Program improvements by source to source transformations, Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, 1976
- NOR72 Nordström, M. FUNSTRUC programdokumentation (in Swedish), Datalogilaboratoriet, Uppsala University, DLU 72/16, 1972
- OSK73 Oskarsson, Ö. Programdokumentation GIP/GUP, Generell in- och utmatning av egenskapslistor (in Swedish), Datalogilaboratoriet, Uppsala University, DLU 73/17, 1973
- RIS73 Risch, T. REMREC - A program for automatic recursion removal in LISP, Datalogilaboratoriet, Uppsala University, DLU 73/24, 1973

- RIS74 Risch, T. PMG - en Program-Manipulator-Generator i LISP (in Swedish), Datalogilaboratoriet, Uppsala University, DLU 73/24, 1973
- SAN71 Sandewall, E. A programming tool for management of a predicate-calculus-oriented data base, Proc. of Second International Joint Conference on Artificial Intelligence, British Computer Society, 1971
- SAN73 Sandewall, E. Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs, Proc. of Third International Conference on Artificial Intelligence, Stanford, Calif, 1973
- SAN76 Sandewall, E. Some observations on conceptual Programming, Informatics Laboratory, Linköpings University, Lith-MAT-R-76-8, 1976
- SAN77 Sandewall, E. Research Methodolgy in Computer Science, Informatics Laboratory, Linköpings University, 1977 (in print)
- SCH72 Schenk, P.B. and Angel, E. A FORTRAN to FORTRAN optimizing compiler, The Computer Journal 16, 4, 1972
- TEI73 Teitelman, W. CLISP - Conversational LISP, Proc. of Third International Joint Conference on Artificial Intelligence, Stanford, Calif, 1973
- TEI74 Teitelman, W. INTERLISP Reference Manual, Xerox, Palo Alto Research Center, Calif, 1974
- URM77 Urmi, J. Private communications.
- WEG75a Wegbreit, B. Property Extraction in Well-Founded Property Sets, IEEE transactions on software engineering, vol SE-1, No. 3, 1975

- WEG75b Wegbreit, B. Mechanical Program Analysis, Com. ACM 18, 9, September 1975
- WEG76 Wegbreit, B. Goal-directed program transformation, Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, 1976
- WIR71 Wirth, N. Program development by stepwise refinement, Comm. ACM 14, 4, April 1971



