GRAAL : A FUNCTIONAL PROGRAMMING SYSTEM WITH UNCURRYFIED
COMBINATORS AND ITS REDUCTION MACHINE

Patrick BELLOT

LITP-CNRS (LA 248), 2 place Jussieu, 75005 PARIS
Université de Metz, Ile du Saulcy, 57045 METZ-CEDEX

Abstract: This work presents a new programming language issued from Functional Programming systems of J.W. Backus. GRAAL is a functional programming language without variables. The basic tools provided for expressing functions are functional forms as in FP systems. The main change is that functional forms are constructed using uncurryfied combinators, that is polyadic combinators. The introduction of a new kind of functional form (user form) allows users of the system to define their own functional forms and to compute them dynamically. Another difference with FP systems is that functions are polyadic. The main consequence of these choices is an efficient reduction machine, implementing the language, which is described here. Independently of efficiency considerations, GRAAL seems to be a realistic programming language.

Résumé: Ce travail présente un nouveau langage de programmation fonctionnelle issu des systèmes FP (Functional Programming systems) de J.W. Backus. GRAAL est un langage de programmation fonctionnelle sans variables. Comme dans les systèmes FP, les formes fonctionnelles sont les outils fondamentaux servant à exprimer les fonctions. La principale différence réside dans le fait que ces formes sont contruites à l'aide de combinateurs décurryfiés, c'est à dire polyadiques. L'introduction d'un nouveau type de forme fonctionnelle (formes utilisateurs) permet à l'utilisateur du système de se définir ses propres formes fonctionnelles et de les construire dynamiquement. Enfin, les fonctions de GRAAL sont polyadiques. Une conséquence de ces choix est l'existence d'une machine de réduction efficace réalisant le langage GRAAL et qui est décrite dans cette article. Indépendamment de ces considérations d'efficacité, GRAAL semble être un langage de programmation à part entière.

This work was partially supported by a grant from ADI (contract 83/695) and by the CNRS GRECO n° 36.

1) Introduction

There is a growing interest in the eighties for functional programming. In this context, FP systems presented by J.W. Backus in 1978 are the starting point of a great diversity of developments.

This article describes GRAAL (General Recursive Applicative and Algorithmic Language), a functional programming language and its virtual reduction machine. The goal of GRAAL is to provide solutions to some imperfections of Backus's FP systems which are shortly described below.

First at all, the main defect of FP is its inability to handle functional arguments and results. In order to obtain a simple semantic for FP, J.W. Backus has restricted the power of FP so that FP systems seem to be some kind of mathematical formalism to express first-order functions.

For a programming stand-point, it is rather disgraceous that users cannot define new functional forms and must use those provided by the implementation.

In order to express its semantic, a FP system needs a metasytem in which it is possible to express high order computations. FFP were choosen by J.W. Backus.

There is a clear confusion in FP between cartesian products and linked lists which are together designed as sequences. When used as an argument, a sequence is a cartesian product. Its components are accessed with projections. The other use of the sequence is classical in fonctionnal programming: a list accessed by its head and its tail.

More than this, the use of a sequence of arguments for each call of function consumes a prohibitive amount of memory since every sequence must be constructed.

In order to poise the balance, it must be pointed out that FP systems seems to be the first usable programming language without variables even if this possibility were known since H.B. Curry and M. Schoenfinkel. The absence of variables allows a clear view on the tasks to perform, a trivial but noticeable example is:

```
#DEF caddr (car o cdr o cdr)
```

An Algebra of Programs is also given with each FP. This algebra permits reasoning on programs in a purely algebraic way as we do in arithmetics. Viewing laws of the algebra as term rewriting rules, [2] designed a system that speeds up FP programs.

The conceptual simplicity of FP has permitted some very nice results such as [4,8] and allows very simple implementations [3].

2) Description of GRAAL

A GRAAL system consists of a set of objects, a set of primitive functions, a set of primitive operators, a set of definitions and the operation of application.

2.1) The set of objects

The set of objects is defined inductively:

- every number is an object (eg: -23, 1062)
 - every symbol is an object (eg: foo, bar)
 - the *undefined*, denoted $\perp$, is an object
 - the *empty list*, denoted $\langle \rangle$, is an object
 - if A and B are objects, the list $\langle A . B \rangle$ is an object
 - if Opf is a symbol and $F_1, \dots, F_n$ are objects, then the *user defined functional form* (Opf $F_1 \dots F_n$) is an object
 - the result of applying a *functional operator* to the good number of arguments is called a *primitive functional form* and is an object.
- The description of functional operators is given below

Every object must be constructed by a finite number of applications of these rules. Because of the intuitive meaning of $\perp$, we must assume that every object constructor is $\perp$ -preserving, that is: every object containing $\perp$ is equal to $\perp$.

2.2) The application

The application of an object F to some objects $X_1, \dots, X_n$ is denoted:

$$(F : X_1 \dots X_n)$$

F is called a function and $X_1, \dots, X_n$ are its arguments. The precise meaning of the application is given below.

2.3) The set of functional operators

Functional operators are special functions denoted by symbols which are to be applied to some arguments in order to get functional forms. For each primitive functional form, there is a functional operator which can compute it.

Let us consider the functional form of composition $(F \circ G)$, there is a function *comp* such that:

$$(\text{comp} : F \ G) = (F \circ G)$$

Primitive functional forms are particular functions depending on parameters. They are given by the operators and by the rule for applying them. The following is the description of some of them.

Notation: We use the notation $\langle X_1 \ X_2 \dots X_n \rangle$ to denote shortly the object: $\langle X_1 \ . \ \langle X_2 \ . \ \langle \dots \ . \ \langle X_n \ . \ \langle \rangle \rangle \dots \rangle \rangle$

2.4) Call-by-value

Because of its simplicity and of its efficiency, the mechanism of *call-by-value* has been chosen for the transmission of arguments to functions. All functions (functional forms or primitives) are *strict*. That is to say that if an argument is $\perp$, the result is $\perp$.

2.5) Description of some primitive functional forms

Primitive functional forms are given by their functional operator and by their rule of application.

Simple composition: $(F \circ G) = (\text{comp} : F \ G)$
 $((F \circ G) : X_1 \dots X_n) = (F : (G : X_1 \dots X_n))$

Generalized composition: $\{F \ G_1 \dots G_n\} = (\text{gcomp} : F \ G_1 \dots G_n)$
 $(\{F \ G_1 \dots G_n\} : X_1 \dots X_m) = (F : Y_1 \dots Y_n)$
 where: $Y_k = (G_k : X_1 \dots X_m)$ for $1 \leq k \leq n$

List construction: $[F_1 \dots F_n] = (\text{cstr} : F_1 \dots F_n)$
 $([F_1 \dots F_n] : X_1 \dots X_m) = \langle Y_1 \dots Y_n \rangle$
 where: $Y_k = (F_k : X_1 \dots X_m)$ for $1 \leq k \leq n$

Conditional form: $(P \longrightarrow F ; G) = (\text{if} : P \ F \ G)$
 $((P \longrightarrow F ; G) : X_1 \dots X_n)$
 $\quad = (G : X_1 \dots X_n)$ if $(P : X_1 \dots X_n) = \langle \rangle$
 $\quad (F : X_1 \dots X_n)$ if $(P : X_1 \dots X_n) \neq \perp$
 $\quad \perp$ otherwise

Constant form: $!X = (\text{cste} : X)$
 $(!X : Y_1 \dots Y_n) = X$

Apply form: $\wedge F = (\text{lapp} : F)$
 $(\wedge F : \langle X_1 \dots X_n \rangle) = (F : X_1 \dots X_n)$

Distribution form: $\&F = (\text{dist} : F)$
 $(\&F : \langle X_1 \dots X_n \rangle) = \langle (F : X_1) \dots (F : X_n) \rangle$

Reduction form: $/F = (\text{redu} : F)$
 $(/F : \langle X_1 \ X_2 \dots X_n \rangle) = (F : X_1 \ (/F : \langle X_2 \dots X_n \rangle))$
 $(/F : \langle X_1 \rangle) = X_1$

Binary to unary form: $(\text{binu} \ F \ Y) = (\text{binu} : F \ Y)$
 $((\text{binu} \ F \ Y) : X) = (F : X \ Y)$

Because of lack of space, it is not possible to give a complete description of the primitive functional form. The other functional forms used in this work are described in the first appendix.

2.6) Primitive functions

The set of primitive functions is issued from Lisp and APL. Primitive functions are denoted using symbols. These are some useful primitive functions given by their rule of application. A more complete list is given in the second appendix.

```
(id : x) = x

(car : <a . b>) = a
(cdr : <a . b>) = b
(cons : a b) = <a . b>

(add : a1 .. an) = a1+...+an
(mul : a1 .. an) = a1*...*an
(sub : a b) = a-b
(div : a b) = a/b if b≠0

(ucar : (Opf P1 .. Pn)) = Opf
(ucdr : (Opf P1 .. Pn)) = <P1 .. Pn>
(ucons : Opf <P1 .. Pn>) = (Opf P1 .. Pn)
```

We can think that every primitive function of classical functional language is included in the set of primitive functions of GRAAL. As was said before, every function is strict.

2.7) The set of definitions

A definition in a GRAAL system is an expression:

```
#DEF    Name    Body
```

where Name is a symbol and Body is any object of the system. Such an expression means that the symbol Name becomes the name of the function Body. If the symbol Name is applied to some arguments, the interpreter replaces it by its body.

2.8) Expressions

Expressions are defined inductively:

- every object is an expression
- if $F, X_1, \dots, X_n$ are expressions, the application of F to $X_1, \dots, X_n$ is denoted $(F : X_1 \dots X_n)$ and is an expression
- every expression is finite (closure rule)

Expressions must be evaluated, that is to say reduced. The reduction of an expression gives a result which is an object. If this result is the undefined (1), we must say that there is no result. By analogy to combinatory calculi, we say that an expression has no normal form if the process of reduction does not stop for this expression.

2.9) Evaluation of expressions

The process of evaluation of an expression is very simple and can be described using the two classical tools *Eval* and *Apply* described in the metalanguage of GRAAL.

EVAL(E)

- 1) if E is an atom, the result is just itself
- 2) if E is a sequence, the result is the sequences of the evaluations of its components
- 3) if E is a functional form (primitive or user defined), the result is the form itself.
- 4) if E is an application then apply the function to the arguments evaluated using *APPLY*.

APPLY(F,X₁,...,X_n)

- 1) if F is an application (G : Y₁ .. Y_m), the result is:
 $APPLY(APPLY(G,Y_1,...,Y_m),X_1,...,X_n)$
- 2) if F is an integer p between 0 and the number of arguments, then the result is the pth argument, otherwise it is 1.
- 3) if F is a symbol denoting a ordinary primitive function then apply its rule to get the result. For instance:
 $APPLY(add,1,2) = 3$
- 4) if F is a symbol denoting a primitive metafunction such as funcall, then apply its rule to get the result. For instance:
 $APPLY(funcall,F,X_1,...,X_n) = APPLY(F,X_1,...,X_n)$
- 5) if F is a symbol serving as name for a function whose body is B, then the result is $APPLY(B,X_1,...,X_n)$
- 6) if F is a user functional form (Opf P₁ .. P_m), then the result is:
 $APPLY(APPLY(Opf,P_1,...,P_m),X_1,...,X_n)$
- 7) if F is a primitive functional form, then apply its rule in order to get the result. For instance, if F is GoH then the result is:
 $APPLY(G,APPLY(H,X_1,...,X_n))$

Remark: the distinction has been made between ordinary functions such as addition (case 3) and metafunctions such as funcall (case 4) because in the first case the result is a value (an object) whereas in the other case the result is a meta-expression containing occurrences of *APPLY* and *EVAL* which must be evaluated in order to get the final result.

3) Programming examples

The examples given in this section are classical. The '~' character marks the beginning of a comment which ends at the end of line.

3.1) Factorial and Fibonacci numbers

```
#DEF    zerop    (binu eq 0)      ~ predicate "equal to zero"

#DEF    fact     (zerop → !1 ; {mul id fact o sub1})

#DEF    less2    (binu le 2)      ~ predicate "less than 2"

#DEF    fibo     (less2 → !1 ;
                  {add fibo fibo o sub1} o sub1
                  )
```

The main thing to remark is that programming in GRAAL means abstraction. That is to say that a programmer must do the abstraction by itself and obtain {mul id fact o sub1} from {mul : x (fac : (sub1 : x))) while Lambda-languages provides the abstraction tools.

3.2) Last component of a sequence and inversing a sequence

```
#DEF    last     (cdr → last o cdr ; car)      ~ recursive

#DEF    last     (car o (while cdr cdr))        ~ iterative

#DEF    reverse  (binu rev2 <>)                ~ reverse with 2 arguments

#DEF    rev2     (1 → {rev2 cdr o 1 [car o 1 . 2]} ;
                  2
                  )
```

Remark: construction form [f . g] can be rewritten as {cons f g}.

3.3) The inner product and matrice product revisited

```
#DEF    ip       (gredu add mul)

#DEF    mp       ({dist1 {distr ip}} 1 trans o 2)
```

The main difference between this version of ip and the one given in [1] is that no memory conses is used in GRAAL for ip. Since the gredu form is primitive, the entire work is done by the interpreter and, consequently, does not consumes no intermediary conses. The only consummation of memory in mp is due to the transposition of the second matrice and to the construction of the result matrice. That is to say that memory is used only when it is explicetely programmed.

3.4) A extensional fixed-point operator

This is the first example of a user defined functional form. Let us compute a form `fix` such that `(fix f)` is a extensional fixed-point of `f`, that is to say:

$$((\text{fix } f) : x_1 \dots x_n) = ((f : (\text{fix } f)) : x_1 \dots x_n)$$

Since we know (definition of `APPLY`, case 6) that:

$$((\text{fix } f) : x_1 \dots x_n) = ((\text{fix} : f) : x_1 \dots x_n)$$

we must have:

$$\begin{aligned} (\text{fix} : f) &= (f : (\text{fix } f)) \\ &= (f : (\text{\$fix id} : f)) \\ &= (\text{funcall} : f (\text{\$fix id} : f)) \\ &= (\text{funcall id } (\text{\$fix id})) : f \end{aligned}$$

thus we must define:

```
#DEF    fix      (funcall id  {\$fix id})
```

I conjecture that this form can be used to suppress recursivity in GRAAL. In the case of unary functions and continuous functionals, this result has been established in [4]. That is to say that the function memory can be suppressed since all recursivities can be factorized into one which can be incorporated in the interpreter.

3.5) A typing form

Since GRAAL allows users to untype primitive functions (in order to exchange more efficiency against security), it is rather interesting to have a typing form `(type p error f)` such that:

$$((\text{type } p \text{ error } f) : x_1 \dots x_n) = ((p \longrightarrow f ; \text{error}) : x_1 \dots x_n)$$

where `f` is the function to type, `p` is a predicate which must be verified by the arguments and `error` is any error recovery mechanism. Because of the rule of application of a user defined form, we must have:

$$(\text{type} : p \text{ error } f) = (p \longrightarrow f ; \text{error}) = (\text{if} : p f \text{ error})$$

thus, the definition of form is:

```
#DEF    type      (if 1 3 2)
```

and a definition of a typed factorial function would be:

```
#DEF    tfac      (type (and integerp (binu ge 0))
                    error
                    fact
                    )
```

since the argument must be an positive integer.

4) A fast reduction machine for Graal

One of the main advantages of the absence of variables is the ability to design very efficient implementation for the language. It is known that such implementations also exists for lambda-languages such as ML (see [6] and [7]), but ML is a strongly typed language and is compiled thus it has lost a few of the charm of applicative languages.

The reduction machine presented in this work is designed for interpreting the GRAAL language. No compilation is done. This machine runs on a classical Von Neumann machine and is very similar to a Lisp interpreter. In Lisp, a large part of execution time is spent while saving and restoring values of variables. Since there is no variables in GRAAL, it is clearly an evidence that a GRAAL interpreter will be faster than a Lisp interpreter.

Another reason of efficiency of GRAAL machine is its conceptual simplicity. There is no intricate operation to realize. Even the detection of tail recursivity and almost-tail recursivity is not needed. All data in the stack at a given time are really needed for the rest of computation.

4.1) Memory organization

Memory is classically divided into four areas: one for each kind of object. These areas called zones are:

- the zone of code which contains the compiled programs of the interpreter.
- the zone of conses which are records of two values: a car and a cdr. conses are used to construct lists, functional forms, applications...
- the zone of symbols which contains attributes of symbols.
- the zone of strings which contains all strings used in the system.
- the zone of stack which contains the stack needed by the interpreter.

4.2) Representation of objects

Objects are represented using words containing pointers to memory. These words must support a flag called a type.

An integer is a word containing an limited integer value and a type INTE specifying that it is an integer.

A list <A.B> is a pointer into the conses zone with the flag CONS. The pointed cons contains representation of A in its car and representation of B in its cdr.

A user functional form ($\text{Opf } P_1 \dots P_n$) is a pointer into the conses zone with the flag USER. The pointed cons contains the representation of Opf in its car and the representation of $\langle P_1 \dots P_n \rangle$ in its cdr.

A primitive functional form is represented by its sequence of parameters with an appropriate flag (for instance COMP for composition).

A symbol is represented by a pointer to the record of its attributes in the symbol table.

4.3) Organization of the stack

If we leave details out, the stack uses two virtual entities: call frames and argument frames.

4.3.a) Argument frame

An argument frame is mainly all the arguments of a function pushed on the stack with a heading specifying that what is under it in the stack is an argument frame and giving its length. This is the general form:

If we know a pointer PA to the argument frame heading, we can recover instantaneously the kth argument by computing its pointer $(PA+k)$.

4.3.b) Call frame

A call frame is used each time an application must be evaluated. It consists simply of three words pushed down on the stack. These words are:

- 1) The representation of the function.
- 2) A pointer to an argument frame.
- 3) A pointer to where the result must be placed.

The general form of a call frame is:

4.3.c) Frame combination

Using the two kinds of frame, an application $(F : X_1 \dots X_n)$ is represented in the stack this way:

Remark 1: An argument frame can be shared by several call frames. This is the case when a composition $\langle F G_1 \dots G_n \rangle$ is applied: arguments are the same for all functions $G_1, \dots, G_n$.

Remark 2: An argument frame is always placed in the stack under the call frames which use it.

Remark 3: Since results of computations are used by other functions, the result address PR is always in the stack.

4.4) The toplevel loop

The toplevel loop examines the top element of the stack. If this element is a heading of call frame, it needs just to skip it (see Remark 2 above). Otherwise we know that the top element of the stack is the function of a call frame.

Associated to each type of object (INTE,CONS,...), there is a procedure which realizes the application of the object and returns to the toplevel.

If we add such a procedure for the ARGV type, the toplevel loop is very simple and looks like:

```
Toplevel loop
  jump to the procedure associated to the
  type of the element at the top of stack.
End
```

Now, we just give some examples of procedures associated to the top of stack element:

4.4.a) The ARGV procedure

This procedure is called when the top of stack is a call frame heading:

```
ARGV procedure
  decrement the stack pointer of the argument count
  plus one (that is enough to skip the argument frame)
  and jump to the toplevel loop.
End
```

4.4.b) The SYMB procedure

This procedure is called when the top of stack element is a symbol:

```
SYMB procedure
  . if the symbol is a primitive function, jump to the
    procedure which realizes the primitive function,
  . if the symbol is the name of a function whose body
    is B, replace the top of stack element by B and
    jump to the toplevel loop,
  . otherwise, this is an error: the symbol is not the
    denotation of a function.
End
```

4.4.c) The INTE procedure

This procedure is called when the top of stack element is an integer, that is an argument selector:

INTE procedure

check that top of stack element is positive and less than the argument count of the argument frame, then take the indicated argument and place it at the result adress.
decrement the stack pointer three time and jump to the toplevel loop.

End

4.4.d) The COMP procedure

Now I just give one example of a primitive form procedure: the composition form. This procedure is applied when the top of stack element is a composition (F o G). The stack looks like:

The COMP procedure must realize a simple transformation of the stack in order to obtain:

When this transformation is done, the COMP procedure makes a jump to the toplevel procedure.

4.5) Tail recursivity

The detection of tail recursivity is a very simple task in GRAAL. A call is tail recursive if its argument frame is just below it. More precisely, the pointer of the argument frame is equal to the top of stack pointer minus three.

In order to handle tail recursivity it is possible to begin the body of each primitive form procedure by a test of the argument frame pointer and execute a different sequence of code if the call is tail recursive. Then we are sure that no obsolete data are kept in the stack but the code of the interpreter is twice longer.

An other manner of doing the same thing is to use a two-stacks system where computations are done alternatively on each stack. The code is considerably reduced and the spirit of the implementation is kept entire.

The better way is a slight modification of this machine where the top of stack call frame and the argument frame are kept into six registers: one for the function, one for the result address, one for each three foremost arguments and one containing the sequence of the rest of arguments. With this modification, the code of the interpreter becomes very small and much more easy to write. Detection of tail recursivity is no more needed.

5) Concluding remarks

This article describes a language and a fast reduction machine for it. Written in the C language on a VAX 11-780 under system UNIX 4.2, the execution time needed to compute the 21st number of Fibonacci (10946) is nearly 1700ms.

GRAAL is not designed to be use as an every day programming language since its syntax is quite unreadable and its use requires good abstraction technics. But GRAAL can be used as an implementation for lambda-languages. A Lisp (with lexical binding) interpreted in GRAAL may have the same execution time as GRAAL itself.

GRAAL allows high order programming in the sense of [4], that is to say: functional arguments and results. Functionals of any order are programmable without lost of efficiency and without problems since there is no variables thus no environment to preserve.

GRAAL has the advantage of being efficient on classical Von Neumann machine and do not requires special architectures. Nevertheless, an implementation of GRAAL on new architectures must be studied since the absence of environment allows a complete sharing of memory between different processors.

6) REFERENCES

- [1] J.W. Backus
Can programming be liberated from Von Neumann style? A functional style and its Algebra of programs., CACM 21, Volume 8, 1978.
- [2] F. Bellegarde
Rewriting systems on FP expressions that reduce the number sequences they yield, Symposium on Lisp and Functional Programming 1984, LNCS , pp 63-73, Austin (Texas), 1984.
- [3] P. Bellot
JYM : un langage de programmation sans variables et ses réalisations, Actes des journées Afcet-Groplan, Bulletin BIGRE+GLOBULE, J. André éditeur, à paraître, Toulouse 1985.
- [4] P. Bellot
High order programming in extended FP, IFIP conference on Functional Programming and Computer Architecture, LNCS 201, pp 65-80, Nancy 1985.
- [5] P. Bellot and B. Robinet
Functional programming with uncurryfied combinators, First Workshop on Reduction Machines, to appear in LNCS, Ustica (Italy), 1985.
- [6] L. Cardelli
Compiling a functional language, Symposium on Lisp and Functional Programming 1984, LNCS, Austin (Texas) 1984.
- [7] G. Cousineau, PL. Curien and M. Mauny
The categorical abstract machine, IFIP conference on Functional Programming and Computer Architecture, LNCS 201, pp 50-64, Nancy 1985.
- [8] J.H. Williams
Formal representation of recursively defined functional programs, Formalization of programming concepts, LNCS 107, Penniscola 1981.

APPENDIX IPrimitive functional forms

Left distribution form: $(\text{distl } f) = (\text{distl} : f)$

$((\text{distl } f) : \langle x_1 \dots x_n \rangle y) = \langle (f : x_1 y) \dots (f : x_n y) \rangle$

Riqth distribution form: $(\text{distr } f) = (\text{distr} : f)$

$((\text{distr } f) : x \langle y_1 \dots y_n \rangle) = \langle (f : x y_1) \dots (f : x y_n) \rangle$

Form construction form: $\$Opf = (\text{form} : Opf)$

$(\$Opf : p_1 \dots p_n) = (Opf p_1 \dots p_n)$

Generalized reduction form: $(\text{gredu } f g) = (\text{gredu} : f g)$

$((\text{gredu } f g) : \langle a_1 . b_1 \rangle \langle a_2 . b_2 \rangle \dots \langle a_m . b_m \rangle)$
 $= (g : a_1 a_2 \dots a_m) \text{ if one of the } b_i \text{ is atomic}$
 $(f : (g : a_1 a_2 \dots a_m) ((\text{gredu } f g) : b_1 b_2 \dots b_m)) \text{ otherwise}$

Generalized distribution form: $\%f = (\text{gdist} : f)$

$(\%f : \langle a_1 . b_1 \rangle \langle a_2 . b_2 \rangle \dots \langle a_n . b_n \rangle)$
 $= \langle (f : a_1 a_2 \dots a_n) \rangle \text{ if one of the } b_i \text{ is atomic}$
 $\langle (f : a_1 a_2 \dots a_n) . (\%f : b_1 b_2 \dots b_n) \rangle \text{ otherwise}$

And form: $(\text{and } f_1 \dots f_n) = (\text{and} : f_1 \dots f_n)$

$((\text{and } f_1) : x_1 \dots x_m) = (f_1 : x_1 \dots x_m)$

$((\text{and } f_1 f_2 \dots f_n) : x_1 \dots x_m)$
 $= \langle \rangle \text{ if } (f_1 : x_1 \dots x_m) = \langle \rangle$
 $((\text{and } f_2 \dots f_n) : x_1 \dots x_m) \text{ if } (f_1 : x_1 \dots x_m) \neq \langle \rangle$

Or form: $(\text{or } f_1 \dots f_n) = (\text{or} : f_1 \dots f_n)$

$((\text{or } f_1) : x_1 \dots x_m) = (f_1 : x_1 \dots x_m)$

$((\text{or } f_1 f_2 \dots f_n) : x_1 \dots x_m)$
 $= (f_1 : x_1 \dots x_m) \text{ if } (f_1 : x_1 \dots x_m) \neq \langle \rangle$
 $((\text{or } f_2 \dots f_n) : x_1 \dots x_m) \text{ if } (f_1 : x_1 \dots x_m) = \langle \rangle$

APPENDIX IIPrimitive functions

```

(id : x) = x

(car : <a . b>) = a
(cdr : <a . b>) = b
(cons : a b) = <a . b>
(null : a) = t if a=<>, <> otherwise

(ucar : (Opf p1 .. pn)) = Opf
(ucdr : (Opf p1 .. pn)) = <p1 .. pn>
(ucons : Opf <p1 .. pn>) = (Opf p1 .. pn)

(add : x1 .. xn) = x1+...+xn if x1,...,xn are numbers
(mul : x1 .. xn) = x1*...*xn if x1,...,xn are numbers
(sub : x y) = x-y if x and y are numbers
(div : x y) = x/y if x and y are numbers and y≠0

(sub1 : x) = x-1 if x is a number
(add1 : x) = x+1 if x is a number

(eq : x y) = t if x and y are equal, <> otherwise

(≤ : x y) = x if x and y are numbers and x≤y, <> otherwise
(≥ : x y) = x if x and y are numbers and x≥y, <> otherwise
(< : x y) = x if x and y are numbers and x<y, <> otherwise
(> : x y) = x if x and y are numbers and x>y, <> otherwise

(apply : f <x1 .. xn>) = (f : x1 .. xn)
(funcall : f x1 .. xn) = (f : x1 .. xn)

(trans : <<a1 . b1> <a2 . b2> .. <an . bn>>)
      = <<a1 a2..an>> if one of the bi is atomic
      <<a1 a2..an> . (trans : <b1 b2..bn>>)

```