UPPSALA UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCES
REPORT NR 9
SEPTEMBER, 1967

Ambiguity logic as a basis for an incremental computer

by

Erik Sandewall

The research reported here was partly supported by the Swedish Natural Science Research Council (Dnr 2711-2 ans 6826) and by the Advanced Research Projects Agency of the Office of the U.S. Secretary of Defence (SD-183).

Introduction

This memo describes a LISP program for an incremental computer, and the logical system that is the basis for the program.

By an incremental computer (IC), we mean a (hardware and/or software) device that

(1) is able to accepte facts or statements in some general notation and incorporate them into a data structure;

(2) can be programmed so as to perform some operations automatically, when it receives certain types of facts, or recognizes certain combinations of facts. These automatically performed operations could involve (a) deposit of new facts in the data base. In this case, the IC serves as an automatic inference device; or (b) performing some external operations, e.g. printing on a teletype a report of what it is doing.

Some previously written programs satisfy parts of this definition:

(A) Many question-answering programs (1,3) perform some deductions from given facts at assertion time, rather than at question time. (This is often done in order to detect inconsistencies in given facts.) As incremental computers, such programs are weak in two respects, however. First, they can only operate on facts within a narrow range of discourse, and second, the program that performs the inference is large and complicated. We would prefer to write the program for the IC in a more modular fashion. Each rule should be written on the form "whenever you see this, do that" and incorporated into the IC independently of other rules.

(B) The incremental computer described by Lombardi and Raphael(4) accepts one simple kind of facts: a certain LISP atom has a certain value.

When such a fact is fed in, the computer resumes evaluation of expressions that previously stalled for the lack of value in this atom. These expressions can of course be written with full generality. The restriction lies in the extreme narrowness of the class of facts that this IC will accept.

In the present memo, we shall describe an incremental computer with the following characteristics:

(1) It accepts expressions in a modification of predicate calculus. The modification consists in introducing the ambiguity operator and some related functions. The new symbols supplant the universal quantifier and (to some extent) set theory notation.

(2) Each input expression can be prefixed by a ! (for "the following fact shall be incorporated into the data base"), or a ? (for "can the following statement be deduced from the data base?"). These two cases are handled exactly the same, except in the final step. Mainly, the program looks at the expression and its sub-expressions; performs all inferences that it "comes to think of" (i.e. all inferences given by operators of the type "whenever you see an expression like .... , do ...."). If the prefix is ! , the program flags the expression as true (which may cause further inference, setting truth-values of other expressions); if the prefix is ?, the program checks whether the expression has gotten a flag for truth.

(3) The operators that together form the program <u>for</u> the IC (in contrast to the LISP program that <u>simulates</u> the IC) are in the first hand built for logical inference, but they can use a function <u>evalq</u> which calls the LISP eval. It is therefore possible to "deduce" that some operation outside the realm of logics shall be taken. This is

like of we could write, in ordinary predicate calculus, expressions like

$$(t) \quad (t \in R \supset \text{evalq(print (describe(t))))} \;,$$

meaning that each time the relation $t \in R$ is proved true for some t , we let the LISP eval evaluate "describe(t)" and print it.

(4) Logical inference in the IC can be concerned with real-world objects, mathematical items, etc., but it is also possible to quote expressions already in the IC; to state their properties (e.g. their "significance") and the relations between them; and to make inference on the facts about them. In a certain sense, therefore, the program is potentially capable to "think about its own thinking".

This memo consists of three chapters, which deal with:

(1) the ambiguity logic that underlies the IC;

(2) the LISP program that simulates the IC. Essentially, this is a program that performs automatic inference on a small number of axioms. All other axioms (and all extra-logical operators) can then be expressed as operators that are called and executed by the IC program;

(3) the belief structure character of the IC's data base (the IC flags some expressions as "true according to present knowledge"). We discuss the notation for keeping track of truth-values and some problems which arise when the IC first "believes" that an expression is true and then "changes its mind

One version of the program for the IC is working, but lacks some of the features described in this memo. A later version with all the features is not completely debugged.

Chapter 1

Ambiguity logic

1.1 The basic notation

We shall use the ambiguity operator, mentioned by McCarthy in (2), in the logical system for the IC.

Let us use the question-mark (?) as an infixed, binary operator, and let "$a ? b$" be an ambigous expression whose value is a, or b. We have to make this more precise. When we write "$f(x) = a?b$", we could mean:

(1) The value of $f(x)$ may be a, and it may be b. We do not know (or we do not care to tell).

(2) $f(x)$ has two values. One of them is a; the other is b.

Some axioms hold in either case, viz.

(A1) $a ? a = a$

(A2) $a ? b = b ? a$

(A3) $a ? (b ? c) = (a ? b) ? c$

(A4) $f(a ? b) = f(a) ? f(b)$

The difference between (1) and (2) becomes acute when we are asked whether we want to assume an axiom like

(A?) $a = b \supset a = (b ? c)$

which would permit us to write e.g.

$$2 + 3 = 4 ? 5 .$$

It is more fruitful to consider this as a question what we intend to mean by equality (between ambigous expressions). If accepted, (A?) would violate the transitivity of equality, because we would have

$$3 = 3 ? 4 = 4 = 4 ? 5 = 5 \quad \text{etc.}$$

Therefore, we reject "axiom" (A?) and choose interpretation (2) above.

This also means that we would <u>not</u> write

$$\forall((n), \text{ult}(n) = 0) = T$$

as McCarthy does in (2). (We might instead write $\forall((n), 0 \hookleftarrow \text{ult}(n))$; see below.)

Axiom (A3) makes it natural to omit parentheses in sequences of ?'s , and write e.g. a ? b ? c ? d . We shall say that this is an <u>ambigous expression</u>. It <u>denotes</u> an ambigous object or <u>ambject</u> which <u>contains</u> the cases a,b,c,d, a?b, a?c, ..., a?b?c?d . If we want a strict definition of equality, we can assume a "weak" equality between unambigous expressions to be given, and define a strong equality as follows: a = b iff every unambigous case of a equals some unambigous case of b , and vice versa.

We shall use the symbol Я for the "inconceivable case", so that

$$a \mathbin{?} Я = a$$

If e is an expression, let us write *e for " e contains only itself and Я as cases" or " e is unambigous". We also stipulate $\neg *Я$, and then obtain as an axiom

(A5) $\quad *(a \mathbin{?} b) \supset a = b$

We shall distinguish between <u>regular</u> and <u>dominant</u> functions. A regular function is a function which satisfies axiom (A4) above; a dominant function is one which does not. In particular, ? itself is a regular function. Functions of several arguments can be regular in some of them.

The same distinction is valid for predicates. A regular predicate is one which satisfies

$$p(a \mathbin{?} b) = p(a) \mathbin{?} p(b) .$$

If p(a) = true, p(b) = false, we have p(a?b) = true ? false. Here, ? is a logical connective rather than a function. We obtain a four-valued logic with the values true ($\underline{t}$), false ($\underline{f}$), $\underline{t}$? $\underline{f}$ or sometimes ($\underline{s}$), and Я . $\wedge$, $\vee$, $\neg$ etc. are regular connectives in this logic, and we have e.g.

$$\underline{t} \mathbin{?} \underline{f} = \underline{s}$$
$$\underline{s} \mathbin{?} \underline{t} = (\underline{t} \mathbin{?} \underline{f}) \mathbin{?} \underline{t} = \ldots = \underline{s}$$
$$\underline{s} \wedge \underline{t} = (\underline{t} \mathbin{?} \underline{f}) \wedge \underline{t} = (\underline{t} \wedge \underline{t}) \mathbin{?} (\underline{f} \wedge \underline{t}) = \ldots = \underline{s}$$

It is a trivial task to write down the truth-tables for these four truth-values and the logical connectives $\wedge$, $\vee$, $\neg$, ? . See appendix 1. We may also use a dominant or strong implication, defined in analogy with strong equality: u implies v strongly (written $u \supset v$) if every unambigous case of u implies (weakly, i.e. with regular implication) some unambigous case of v , and every unambigous case of v is implied by some unambigous case of u . It follows that strong implication is transitive, and that

$$\underline{f} \supset \underline{s} \supset \underline{t} \; .$$

The truth-table of strong implication is also given in appendix 1.

A purist might prefer to use different symbols for the function ? and the logical connective ? . However, axioms (A1) - (A5) are valid for both these kinds of ? (if the function symbol f is axiom (A4) is interpreted so that wffs are obtained). We therefore use the same symbol and save duplication of axioms.

By the same philosophy, we consider = and * as dominant predicates and, at the same time, as dominant logical connectives. We have e.g.

$$(\underline{t} = \underline{s}) = \underline{f}$$
$$*(u = v)$$

We need one more function, a dominant function / which is similar to intersection between sets in the sense that a / b shall be an ambigous expression whose cases are the common cases of a and b . This gives us e.g.

$$(a \mathbin{?} b) \mathbin{/} a = a$$

$$f(a \mathbin{/} b) = f(a) \mathbin{/} f(b) \qquad \text{if } f \text{ is regular.}$$

If a and b have no common cases, a/b is Я . Likewise, f(Я) = Я for regular functions f .

Finally, we need one dominant predicate, corresponding to set-inclusion. $a \hookleftarrow b$ shall be <u>t</u> if every case of a is also a case of b (the "vice versa" is not required) and <u>f</u> otherwise. This gives us

$$a \mathbin{/} b \hookleftarrow a \hookleftarrow a \mathbin{?} b$$

$$(a \hookleftarrow b) \wedge (b \hookleftarrow a) \supset (a = b)$$

Of course, / and $\hookleftarrow$ have a second nature as logical connectives.

The ambiguity concept plays us a few tricks. We no longer have

$$(u \vee \neg u)$$

as an axiom. More significant is the impact on the λ-notation, which is the topic of the next section.

1.2 The λ-notation and the ρ-notation

Should we consider a λ-expression as a regular or a dominant function? Let us see what happens when we apply axiom (A4):

$$p(a,a) \mathbin{?} p(b,b) =$$
$$(\lambda(t)p(t,t))(a) \mathbin{?} (\lambda(t)p(t,t))(b) = \text{/axiom (A4) !/}$$
$$(\lambda(t)p(t,t))(a \mathbin{?} b) =$$
$$p(a?b,a?b) =$$
$$p(a,a) \mathbin{?} p(a,b) \mathbin{?} p(b,a) \mathbin{?} p(b,b) .$$

Thus it seems that axiom A4 is not applicable, and that λ-expressions should be considered as dominant functions. However, we may instead question the third equality sign in the example, which relies on the axiom

$$\text{(K)} \quad (\lambda(t)\ F(t))\ (a) = F(a)$$

If we take as a requirement in axiom K that a must be unambigous, then lambda-expressions will indeed be regular functions. We shall permit both cases, and introduce the symbol ρ for the lambda that creates regular functions. Thus every expression $(\rho\ (\ldots)\ \ldots)$ is a regular function, and we have the axiom

$$^{*}a \supset \left[(\rho(t)\ F(t))\ (a) = F(a)\right]$$

λ-expressions, on the other hand, are dominant functions and satisfy axiom K .

This trouble with λ-expressions may seem a nuisance. In fact, it is an asset, because we can use ρ in many cases where a universal quantifier would otherwise be used. Let us work a simple example in ordinary logic notation.

Ex. Let B1, B2, ... Bn be boys, and define the function

father(x) for "the father of x"

and the predicate

admire(x,y) for "x admires y"

The phrase "all boys admire their fathers" is usually written

$\forall((b)\ b \in \text{B1n} \supset \text{admire}(b, \text{father}(b)))$

where B1n = {B1, B2, ... Bn} .

With the ρ-notation, we can instead write

(ρ (b) admire(b,father(b))) (boy)

where boy = B1 ? B2 ? ... Bn .

This works as follows. Let A be the regular predicate

(ρ (b) admire(b,father(b)))

Our axiom states that A(B1 ? B2 ? ... ? Bn) is $\underline{t}$, but as the argument is ambigous, we cannot substitute it into the definition of A . But according to regularity, we have instead

A(B2) $\leftarrow\!\!\urcorner$ A(B1 ? B2 ? ... ? Bn) = $\underline{t}$, and through application of a sequence of obvious axioms, we can conclude

A(B2) = $\underline{t}$

As we also have *B2 , we can now use the axiom for ρ and conclude

admire(B2, father(B2))

In actual use of this notation, we would of course not feel compelled to define "boy" through an enumeration of all possible boys. Instead, we would assume the ambigous object "boy" and use the function A each time we encountered an object Bk with the properties Bk $\leftarrow\!\!\urcorner$ boy, *Bk .

1.3 The let-notation

The lambda-notation (and by consequence, the ro-notation) often yields expressions that are difficult to read, especially when the lambdas are nested. The let-notation, originally proposed by Landin /3/, is an equivalent notation with increased legibility.

A few examples will make the notation clear.

Ex. 1 lambda-notation: (λ(x) f(x,x)) (a)

let-notation: *let* x *be* a *in* f(x,x)

Ex. 2 lambda-notation: (λ(x,y) f(x,y) + f(y,x))(a,b)

let-notation: *let* x *be* a, y *be* b *in* f(x,y) + f(y,x)

Ex. 3 lambda-notation: (λ(x),(λ(y,z) f(y,z) + f(z,y))(f(x,x),g(x)))(g(a))

let-notation: *let* x *be* g(a) *in*

let y *be* f(x,x), z *be* g(x) *in* f(y,z) + f(z,y)

We shall extend the notation to ro-expressions, but distinguish them by writing *be a* or (synonymously) *be an* instead of *be* .

Ex. 4 ro-notation: the example above

let-notation: *let* b *be a* boy *in* admire(b, father(b))

Chapter 2

Ambeval - an eval function for ambiguity logic and ro-expressions

2.1 Use of ambeval

Our program for the incremental computer consists of a LISP function ambeval, which evaluates expressions in the ambiguity logic and takes proper care of ro-expressions; a substantial number of functions that are directly or indirectly called by ambeval; and a small number of "surface functions", which facilitate the use of ambeval in time-sharing mode. This is the program that simulates the IC. The system is written for PDP-6 LISP. Besides, we have written a good number of expressions (for ambeval), whose leading function is a ro-expression. When ambeval evaluates these expressions, it is made to act as an IC with the expressions as independent operators. Together, they form the program for the IC.

In this section, we shall demonstrate the use of the system through the surface functions (e.g. ?, !) . Latter sections explain ambeval and its subfunctions. If possible, the reader should re-do the examples on a computer.

After the ambeval system has been read in and initialized (see separate operating instructions), we start giving it facts, i.e. S-expressions preceeded by an exclamation mark. Object symbols and functions need not be declared before use. Until declared otherwise, each symbol is automatically assumed to stand for an ambject ≠ Я , which is not necessarily unambigous. Examples of use (computer input indented):

```
   (! IMPLIES (OR LOG1 LOG2) (AND LOG3 LOG4 LOG5))
OK
   (! . LOG2)
OK
```

We then pose questions similarily, but with ? instead of ! . This is not the question mark used for the ambiguity operator! The system answers

with 'NIL' if no proof of the formula can be found
with "(BECAUSE ...)" if the formula can be proved from something previously asserted, and with "(BECAUSE)" if the formula has previously been given as an assumption (with !) . Examples:

```
  (? . LOG4)
(BECAUSE (AND LOG3 LOG4 LOG5))
  (? AND LOG3 LOG4 LOG5)
(BECAUSE (OR LOG1 LOG2) (IMPLIES (OR LOG1 LOG2) (AND LOG3 LOG4 LOG5)) )
  (? OR LOG1 LOG2)
(BECAUSE LOG2)
  (? . LOG2)
(BECAUSE)
```

There exist operators that perform (incomplete) inference from the following ambeval functions:

AND, OR, IMPLIES

AMB	(for the ambiguity operator ?),
SLASH	(for /),
ISCASE	(for ↤)
STAR	(for the function * for unambiguity)
ISEQUAL	(for equality and logical identity).

All functions are prefixed. AND, OR, AMB, and SLASH can take an arbitrary number of arguments.

Operators for handling logical NOT, and for functions and relations on sets (e.g. an operator to deduce $a \subset c$ from $a \subset b$, $b \subset c$) do not yet (July 28) work right.

Some time-consuming inferences take place only if certain expressions are declared "interesting" or "significant". This is done by prefixing '>> n' instead of ! or ?. n is an estimate of the significance of the expression;

an integer between 1 and 6, where 6 stands for the highest significance. In the present system, n governs only in what order inferences are to be made, but even inferences on level 1 are performed, sooner or later. When some interrupt device has been introduced into the program, expressions with low significance may go completely unprocessed.

It is easy to change the range of the significance quantity n , if six steps is considered too crude.

Examples of use:

```
  (! ISCASE OBB1 OBB2)
OK
  (! STAR OBB2)
OK
  (>> 4 . OBB2)
YES
  (>> 4 ISEQUAL OBB1 OBB2)
YES
  (? ISEQUAL OBB1 OBB2)
(BECAUSE (ISCASE OBB1 OBB2) (STAR OBB2))
```

In a more practical system, we would of course prefer to have heuristic routines which assign these significance quantities automatically.

2.2 Handling of the functions $\hookleftarrow$ and $*$, and of ro-expressions.

The key ideas in the function evaluator, ambeval, are

(1) Ambeval constructs one property-list for each expression that it is given, and for each sub-expression. For example, when ambeval evaluates (IMPLIES (OR LOG1 LOG2) LOG3), it introduces one property-list for each of the following expressions:

    LOG1
    LOG2
    (OR LOG1 LOG2)
    LOG3
    (IMPLIES (OR LOG1 LOG2) LOG3)

On the property-list for an expression a , ambeval stores e.g. the truth-value of a ; a flag for the truth of *a ; references to property lists for ambjects b such that a $\hookleftarrow$ b or b $\hookleftarrow$ a ; etc. The exact format of the property-list is described below.

Ambeval never introduces several property-lists for one expression, even if it is evaluated repeatedly.

(2) Suppose

(21) A = [ρ (x1 x2 ... xn) expression]

(22) ambeval has evaluated A(y1 y2 ... yn)

(23) the facts z1 $\hookleftarrow$ y1, *z1,
z2 $\hookleftarrow$ y2, *z2, ...
zn $\hookleftarrow$ yn, *zn have been incorporated into ambeval's data structure;

Ambeval shall then proceed to do

(24) evaluate [λ(x1 x2 ... xn) expression] (z1 z2 ... zn)

(25) incorporate the fact

[λ(x1 x2 ... xn) expression](z1 z2 ... zn) $\hookleftarrow$ A(y1 y2 ... yn)

into the data structure.

Ex. 1. Ambeval evaluates

[ρ (x1) evalq(print(describe(x1)))] (boy)

and introduces a property-list for this whole expression. Moreover, ambeval has previously obtained or it later obtains

b $\hookleftarrow$ boy

*b

Ambeval then evaluates

evalq(print(describe(b)))

This is intended to make the LISP system print out a description of the object b . The function evalq always evaluates into Я , so in step 25, ambeval makes

Я $\hookleftarrow$ [ρ (x1) evalq(print(describe(x1)))] (boy)

which is trivially true.

Ex. 2. Ambeval evaluates

[ρ (x1 x2) R(x1 x2)] (s1 s2)

and puts a label for truth on the property-list for this expression. Also, ambeval obtains

v1 $\hookleftarrow$ s1, *v1, v2 $\hookleftarrow$ s2, *v2.

It then evaluates (i.e. introduces a property-list for)

R(v1 v2)

and does

$R(v1\ v2) \hookleftarrow [\rho\ (x1\ x2)\ R(x1\ x2)]\ (s1\ s2)$

Suitably programmed, ambeval may be able to deduce

$R(v1\ v2) \neq$ Я

and then to proceed to

$R(v1\ v2) = \underline{t}$

The ro-expression therefore says "each s1 stands in the relation R to each s2 ".

If ambeval had evaluated R(s1 s2) /instead of the expression with the ro-expression in it/, and labeled it for truth, then ambeval would not automatically evaluate R(v1 v2) under the assumptions above. However, if we instruct it to evaluate R(v1 v2) , ambeval might still label this expression for truth, automatically.

(3) The order in which the IC is given the premises: (a) the expression A(y1 ... yn) ; (b) a fact $zi \hookleftarrow yi$; (c) a fact *zi ; shall be immaterial. Therefore, each of the operations (a)-(c) should perform two things:

(31) trigger ro-into-lambda applications that now have all their premises satisfied;

(32) deposit themselves in the data base, to be available in step (31) of further evaluations.

For example, each time it does *V , ambeval shall

(31')look up all U such that $V \hookleftarrow U$ has been stored away in some previous step (32); and all the ro-expressions that have previously been applied to all those U ; and then convert the ro-expressions into lambda-expressions and apply them to V .

(32') put a flag on some property-list associated with V , indicating that V is unambigous. This flag is used next time we state $V \hookleftarrow UU$ for some UU , or apply some operator on some U such that $V \hookleftarrow U$.

In the rest of this section, we shall describe the organization of the property-lists. Each property-list is on the form (AMBJECT (A1 . P1) (A2 . P2) ... (An . Pn)) , where

AMBJECT	is a marker which serves the same purpose as <u>car</u> of a LISP atom;
Ai	are attributes; and
Pi	are corresponding properties.

In other words, the property-list is organized like the association-lists of conventional LISP. Every such property-list describes an ambject. For convenience, we shall often call the property-list itself an ambject.

The following are some of the attributes that are used:

ENAME The corresponding property is an atom, viz. the external name of the ambject. In the above example, the ambject for 'LOG1' has the ENAME property LOG1, and similarly for LOG2 and LOG3. The ambjects for the non-atomic sub-expressions (like '(OR LOG1 LOG2)') do not have an ENAME property.

MEAN This gives the meaning of an ambject for a non-atomic sub-expression. For example, the MEAN property of the ambject for '(IMPLIES (OR LOG1 LOG2) LOG3)' is a list (a1 a2 a3), where a1 is the ambject whose ENAME is 'IMPLIES' (notice that we have ambjects for functions and relations as well!)

a2 is an ambject whose MEAN is derived from the expression '(OR LOG1 LOG2)'

a3 is the ambject whose ENAME is LOG3.

We shall use a broken underscore to form a name for an ambject from 'its' formula. Thus LOG1 is the ambject whose ENAME is 'LOG1', and (OR LOG1 LOG2) , is the ambject whose MEAN is a list (OR LOG1 LOG2) .

BECAUSE This property should only appear on ambjects that can carry a truth-value. If we believe that the expression for the ambject has the value true (out of the four possible truth-values), then the BECAUSE property is a list of other ambjects which also are taken to be true (i.e. which have a BECAUSE property) and which together imply the present ambject. Clearly, the references must either go in circles, or end in ambjects which have NIL as a BECAUSE property, i.e. which look like

```
(AMBJECT ... (BECAUSE) ... )
```

If we do not have any reasons to believe that an ambject has the value true, then it does not have any BECAUSE property at all.

The same belief structure organization is used for the references to the cases of an ambject and to the ambjects which have a given ambject as a case; and also for the flag which indicates that an ambject is (believed to be) unambigous. In each of these cases, there occurs a list of ambjects which possess a BECAUSE property, and which form the motivation for the reference or flag.

STAR The value of the STAR property is a list of reasons for the unambiguity of the ambject. For example, if we do

```
(! STAR OBB1)
```

the system will put the property (BECAUSE) on the ambject (STAR OBB1), and the property (STAR (STAR OBB1)) on the ambject OBB1 .

CASES Consider an ambject

```
     (AMBJECT ... (ENAME . U) ... (CASES . lista) ... )
```

lista is a new association-list

```
     ((V1 . lis1)(V2 . lis2) ... (Vk . lisk)) .
```

For each j , the system believes Vj ↞ U with the motivation of the ambjects on the list lisj. For example, if we do

```
     (! ISCASE VV UU)
     (! ISCASE WW VV)
```

then the ambject UU will look like

```
     (AMBJECT ...
               (ENAME . UU)
               ...
               (CASES  ( VV (ISCASE VV UU) )
                       ( WW (ISCASE VV UU) (ISCASE WW VV) )
                          ... )
               ... )
```

TYPES This property is organized like CASES, but gives the reverse reference.

OPERS Let A be a ro-expression, for which we have evaluated (A X1 X2 ... Xn) . Ambeval creates ambjects for the Xi, and for the whole expression (A ... Xn) , but not for A itself.

The OPERS property of an ambject C is a list of ambjects for expressions, whose leading function is a ro-expression, and which include the ambject C among their highest-level arguments.

Several other attributes will be introduced later.

The function ambeval is similar to (and was inspired by) the function evalquotel for the incremental computer of Lombardi and Raphael /4/. In their incremental computer, evaluation of lambda-expressions is postponed whenever some arguments are not yet defined; in ambeval, similar postponement takes place when some argument is ambigous.

2.3 Functions for looking at ambjects.

As we saw, ambjects are association-lists, where the values are often lists of ambjects. Because of circularities in the structure, it is rarely possible to print out an ambject as an S-expression. Among the "surface functions" that facilitate work with the system, there are therefore some functions that enable us to print out some representations of ambjects. We have the functions

usenames which is the inverse of the broken underscore introduced above. usenames takes an ambject and reconstitutes the expression that created it, by working down threough MEAN properties until it finds ENAME properties.

lookval Let A be an ambject. '(LOOKVAL A PROP)' finds the PROP property of A and prints out usenames of it. lookval quotes its second argument.

Example: the surface function ?, used in questions, could be defined as

```
(DEFPROP  ?
  (LAMBDA (A)  (LOOKVAL (AMBEVAL A) BECAUSE))
 FEXPR)
```

Moreover, when an expression (! F A1 A2 ... An) is evaluated, the constant != is set to the ambject (F A1 A2 ... An) . Clearly, != can not be printed as it is, but it can be used by usenames, lookval, and ambeval. The other surface functions set the constants ?= viz. >>= in a similar fashion.

Examples of use:

```
  (! . LOG1)
OK
  (LOOKVAL != BECAUSE)
(BECAUSE)
  (? IMPLIES != (AND LOG6 LOG7))
NIL
  (USENAMES ?=)
(IMPLIES LOG1 (AND LOG6 LOG7))
  (DO (SETQ P1 !=))
DONE      (Do is an EXPR which evaluates its only argument and
           returns DONE. We could not have done simply (SETQ P1 !=)
           because of the printout).
  (! IMPLIES != LOG9)
OK
  (LOOKVAL != MEAN)
(MEAN IMPLIES LOG1 LOG9)
    (CAR ?=)
AMBJECT
   (USENAMES (CDR ?=))
((MEAN IMPLIES LOG1 (AND LOG6 LOG7)) (BECAUSE) ... )
  /this is a good way of looking at a whole ambject/.
```

If you want to look at an ambject that is on the property-lists of your present ambject, it is best to reevaluate it (with the prefix ?, which does not set any truth-values). Alternatively, use assoc.

2.4 Quoting of expressions. Automatic inference. Heuristics.

When ambeval evaluates an expression (F A1 A2 ... An) /where F is not a ro-expression or a lambda-expression/, which it has not seen before, it actually creates two new ambjects. One is (F A1 A2 ... An) as discussed before. The other one looks like

(AMBJECT ... (QUOTE (F A1 A2 ... An)) (STAR) (TYPES (F) ...))

and is denoted (' F A1 A2 ... An) (i.e. usenames prints it with the ', and we can describe it to ambeval with ').

Notice that (' F A1 A2 ... An) $\hookleftarrow$ F . In other words, each function (predicate, connective) symbol stands for an ambject, which has as cases all quoted expressions that have this function (etc.) in the leading position. One of the basic tasks of ambeval is to perform this case-inclusion automatically for each new expression, and to enable operators previously applied to the function symbol, to operate on the quoted expression. This is the basis for automatic inference in the IC.

We stated above that the IC performs inferences by "looking at expressions" and "coming to think of conclusions". More precisely, inference rules are often written on the form

" let pp be a p in (expression) " ,

where p is a function symbol and pp therefore, a quoted expression.

Example 1. If we do (! AND LOG1 LOG2) , the IC should conclude that LOG1 and LOG2 are true, and put (AND LOG1 LOG2) as the single BECAUSE reason. An operator that performs this could look like

```
"let i2 be an and in evalq(
  if assoc(BECAUSE, cdr(ambofxpr(i2)) )
    then mapcar(/λ(r) putval(r,
                               list(ambofxpr(i2))
                               BECAUSE) /
                cdr(xpr(i2)) )
  else nil)", where
putval(r,p,a) puts the property p under the attribute a in the
                        ambject r;
ambofxpr(i) = cadr(assoc(QUOTE, cdr(i)))
xpr(i) =      cdr(assoc(MEAN, ambofxpr(i))).
```

Example: if i = (' OR LOG1 LOG2 LOG3), then
ambofxpr(i) = (OR LOG1 LOG2 LOG3), and
xpr(i) = (OR LOG1 LOG2 LOG3)

The actual operator and### that we use, enables further inference from the fact that the arguments of AND have been set true, and is therefore slightly more involved.

Example 2. The rule "if a admires b , then a is dependent on b " could in predicate calculus be expressed as

$$((a)(b),\ \text{admire}(a,b) \supset \text{dependent}(a,b))\ .$$

For the IC the same rule could perhaps be written as

```
let a be an admire in dependent( evalq(argument(1,a)),
                                 evalq(argument(2,a)) ),
```

where argument(n,e) is a lisp expression that evaluates into the n+1'th member of the list xpr[e].

Notice that this operator gives (DEPENDENT ... ...) to ambeval. This causes the evaluation of

(' DEPENDENT ... ...) $\hookleftarrow$ DEPENDENT

which triggers all operators previously applied to DEPENDENT. In this way, chains of inference may arise.

Operators of this character contain more information than the mere axiom they express: they also state when the axiom shall be applied. In the above examples, the axiom is applied, together with modus ponens, each time a specialization of the left-hand side of the implication in the axiom, is given to ambeval. However, we can do more complicated things.

Example 3. Let signif4 and signif5 be ambjects that have quoted expressions as cases. (Being a case of signif4 or signif5 could mean that the expression is particularly significant to e.g. a given question). Suppose we want to apply the axiom of example 2, only to expressions (DEPENDENT ... ...) which have in this way been termed significant. We would re-write the operator as

```
let a be an admire / (signif4 ? signif5) in
     dependent( evalq (argument(1,a)),
                evalq (argument(2,a)) )
```

This operator is triggered whenever we give ambeval an expression (ADMIRE A B) and then do

```
(! ISCASE (' ADMIRE A B) SIGNIF4)
```

or the corresponding for signif5. - It is necessary, of course, that we have some other operator which performs the inference "if a $\hookleftarrow$ b and a $\hookleftarrow$ c then a $\hookleftarrow$ b/c".

In the above example, how is the expression (DEPENDENT A B) set true? (We said that ambjects are set true by the prefix !, which is a surface function, and which can not easily be used inside ro-expressions) This works like on page 18. In other words, the above let-expression translates into an expression whose leading function is a ro-expression. That expression gets its own ambject R , and it is set true by a ! which is prefixed to the expression when it is first put into the IC. Later, we apply the ro-expression to various cases of the ambject ADMIRE, and each time, we obtain an expression (DEPENDENT ... ...) /notice that we do not obtain back the expression (' DEPENDENT ... ...) ! /which is of course a case of R . But R was declared true, and by specialization we obtain therefore the truth of the ambject (DEPENDENT ... ...) .

Actually, the operators in example 2 and 3 are not completely correct, because they do not check whether the given expression (ADMIRE A B) is true or not. This is managed by the pseudo-function RECOGN, described in the next section.

2.5 Order of evaluation of ro- expressions.

As evaluation of expressions with $\star$ and $\hookleftarrow$ means stating a fact and making the conclusions, ambeval has to account for the problems connected with deep trees of conclusions. Such trees occur e.g. if ro-expressions contain expressions a $\hookleftarrow$ b : evaluation of such a ro-expression may trigger the evaluation of many other ro-expressions referenced by b , which in their turn may trigger new ro-expressions. This may take us into very deep deductions and even into infinite loops.

The order of evaluation of such expressions can then be selected in several ways:

(1) <u>depth-first</u>. If evaluation of expression e triggers expressions f1, f2, ... fk, we first evaluate f1 and all its consequences to the end of the tree; and only then start to evaluate f2 .

(2) <u>Queueing</u>. We keep a queue of expressions that are to be evaluated. If e triggers f1, f2, ... fk, these operators are put at the end of the queue, and the evaluation of all is postponed until the expressions before them in the queue have been handled. When we reach f1 , we put the expressions that it triggers at the end of the queue; proceed to f2 ; etc.

(3) <u>Queueing with priority</u>. This is like strategy (2), except that expressions which are deemed particularly significant are permitted to step into the queue at some point other than its end. The assignment of priority values to ro-expressions and to their arguments could be modified according to previous experience.

Alternative (1) is weak, and (3) seems to be a worthwhile extension of (2). We have therefore selected alternative (3).

2.6 Implementation of the / function. The pseudo-functions recogn and someother.

Let A be an expression (ro (x y) ...) , and suppose that the IC has evaluated A(c,d) . For each ci $\hookleftarrow$ c such that *ci , and for each dj $\hookleftarrow$ d such that *dj , the IC should do A(ci, dj) $\hookleftarrow$ A(c,d) . This is possible, since ambeval introduces an ambject for A(c,d) .

For each *new* unambigous ci $\hookleftarrow$ c that we add after the evaluation of A(c,d) , the IC should therefore combine this ci with *every* case of d , and evaluate A anew. It should not re-evaluate A for the old cases of c . In the general case, when we have ro-expressions with an arbitrary number of variables, and add one new case to some argument, we should get all combinations of all cases of all other arguments, but only the new case in the affected argument.

This is used in the implementation of the rule

$$(a \hookleftarrow b) \wedge (a \hookleftarrow c) \supset (a \hookleftarrow b/c) .$$

For each expression with SLASH that ambeval sees, an operator *creates* a new operator with the arguments of the slash-expression as arguments. The new operator, when applied to unpambigous cases, compares the arguments and checks if they are all identical. If they are, the argument is set as a case of the slash-expression; otherwise, nothing happens.

For example, if we create an expression (SLASH AI1 AI2 AI3) , we get a new operator (OP AI1 AI2 AI3) . Suppose unambigous AA has previously been declared a case of AI1 and of AI3 , and is now declared a case of AI2 . OP is applied to every case of AI1 , to the new case of AI2 , and to every case of AI3 , in all possible combinations. In one of these cases, the case of AI1 and the case of AI3 happen to be identical to the new case of AI2 . Therefore, OP sets the new case of AI2 as a case of (SLASH AI1 AI2 AI3) . If the new case of AI2 is either, not a case of AI1, or not a case of AI3, every combination will fail, and the new case does not become a case of (SLASH AI1 AI2 AI3) .

This method is exceedingly slow, but quite convenient to program. Moreover, it enables us to write ambjects that actively recognize their cases. If ppp is a LISP predicate with one argument, ambeval understands recogn(ppp) as an ambject which has as cases, all ambjects r that satisfy ppp(r) . However, an expression recogn(ppp) may <u>only</u> appear as an argument in SLASH-expressions. If we introduce (SLASH AI1 AI2 (RECOGN PPP)) , the specially created operator will compare new cases of AI2 to each case of AI1 (and vice versa, of course), and if the cases are identical, check the case with ppp . If ppp of the case gives a true value, the new case of AI2 becomes a case of the SLASH-expression.

The pseudo-function recogn (pseudo-function, because ambeval handles the function recogn as an exceptional case) can be used for the 'inference-from-true-facts-only' problem of the previous section. Suppose we want to apply the rule "if a admires b, a is dependent on b " to all true expressions (ADMIRE A B) that have been declared significant. We then write

```
    let a be an admire / (signif4 ? signif5) / recogn(expriztrue)
    in dependent( evalq (argument(1,a)), evalq(argument(2,a)))
```

and define expriztrue as a LISP function which checks if the QUOTE property of its argument has a BECAUSE property. (It is in fact a little bit more complicated than this, because the assumptions of the BECAUSE property for (ADMIRE A B) shall be included in the BECAUSE property for (DEPENDENT A B)).

One SLASH-expression may contain several RECOGN-expressions, but they must all stand after the conventional arguments.

One small problem should be mentioned. For technical reasons in the implementation of the application of stored ro-expressions to new cases, we can <u>not</u> successfully write A(x,x) , where A is a ro-expression. If we evaluate the above expression and then do x4 $\hookleftarrow$ x , ambeval will apply A to the arguments (x4,x4) , but it will <u>not</u> combine x4 in one argument position with other cases of x in the other argument. If we want such expressions, we have to use the pseudo-function SOMEOTHER and write e.g.

```
A(x, someother(x))
```

/which combines the new case x4 in the first argument, with every case, including x4 <u>and</u> the old cases, in the second argument/, and

```
A(someother(x), x)
```

/which does exactly the opposite/.

Chapter 3

Handling of truth-values

3.1 Reasons for storing reasons.

Several attributes (i.e. BECAUSE, STAR, TYPES, CASES, EQUALS) assume the corresponding properties to contain lists of "motivations" or "reasons". It would have been possible to construct a simpler ambeval whose data structure does not contain such reasons. We could e.g. put simple flags (in the LISP sense of the word) for true and for unambigous ambjects. The additional effort and storage required to take care of reasons pays off in the following ways:

(1) The system can "make guesses": it can "deduce" that a certain assertion is probably true, and continue to make conclusions from this guess. If the guess later proves mistaken, the conclusions must be annihilated, and this is possible only if the line of reasoning is stored.

(2) The system can find out how how a given conclusion was arrived at. This is potentially important for heuristic purposes (because we may desire to give increased priority to a fact that proved useful) and for proof construction by analogy (where, given a theorem to prove, we find a similar, known theorem and see how it was proved).

The purpose of this chapter is to explain how indications of reason are manipulated by ambeval, and how they can be used. There is also one section about circular reasoning ("C holds, because B holds, because A holds, because C holds, because ..."). Such circles arise very easily in the data structure when "guesses" are taken back.

3.2 Properties and routines for handling truth-values.

The list of attributes in section 2.2 was incomplete. We shall now extend the list.

DNF This property, like the BECAUSE property, can only appear on ambjects that have a truth-value, but unlike BECAUSE, DNF can appear on ambjects with any truth-value (not only those with truth-value t). The DNF property of an ambject a is essentially a logical expression e on disjunctive normal form, such that e ⊃ a. The logical connectives are left out in e, which is therefore just a list (C1 C2 ... Ci ... Cm) of lists (Ci1 Ci2 ... Cij ... Cin) of ambjects.

For example, suppose an ambject AAAA has the DNF property

((L1 L2)(L4)(L6 L5 L3)(L1 L7)) (*)

This means that

$(L1 \wedge L2 \supset AAAA)$

$(L4 \supset AAAA)$

$(L6 \wedge L5 \wedge L3 \supset AAAA)$

$(L1 \wedge L7 \supset AAAA)$.

In other words, (L1 L2) can appear as a possible BECAUSE property for AAAA, if only L1 and L2 have BECAUSE properties;

(*) Notice again that L1 is a LISP atom; that L1 is the ambject whose ENAME , property is L1; and that the DNF property above therefore is a list of lists of ambjects.

(L4) is another possible BECAUSE property; etc.

HELPSIMPLY This property contains references in the reverse direction from DNF. For example, if the ambject AAAA has a DNF property as above, then each of the ambjects L1, L2, L4, L6, L5, L3, L7 has a HELPSIMPLY property, which is a list and has the ambject AAAA as one of its members. If it exists, the HELPSIMPLY property of a given ambject is always a simple list of ambjects (not a list of lists, like the DNF property), and it consists of those ambjects that have the given ambject on their DNF property.

There are essentially two operations that have to be performed on these properties:

(1) "Let new = (n1 n2 --- nk) be a list of ambjects, and let a be an ambject. Incorporate into the data structure, the fact $n1 \wedge n2 \wedge \ldots \wedge nk \supset a$"
The following steps must be taken:

(1a) Look up the DNF property of a, if it exists. If new is a superset of some member of the DNF, then return, otherwise cons new to the property; add a to the HELPSIMPLY property of each member of new; and continue.

(1b) If a has a BECAUSE property, then return.(*) If some member of the list new does not have a BECAUSE property (i.e. is not believed to be true), then return. Otherwise, put new as the BECAUSE property of a, and continue.

(*) For technical reasons the program often proceeds to step 1c instead.

(1c) The system now has concluded that a is true, and this is something it did not know before. To update all ambjects that have a on their DNF, perform the following routine recursively:

(1c1) Consider each member a' of the HELPSIMPLY property of a.

(1c2) If a' has a BECAUSE property, return. Otherwise, try to find some member rrrq of the DNF property of a' each of whose members has a BECAUSE property. If we find rrrq (in which case we can be sure that it contains a), put it as a BECAUSE property of a', and start again from step (1c1) with a' instead of a. Otherwise, return.

In this process, each time an ambject a is set true (by giving it a BECAUSE property), those ro-expressions that only operate on true ambjects are given a chance to discover a. This can e.g. be accomplished by evaluating
a ↢ recogn(expriztrue).

In the LISP program for ambeval, steps (1a)-(1c) are taken if we do "putreason (new,a)". Step (1c2) is performed by "rechecktrue (a')".

(2) The ambject a has a BECAUSE property. This property is incorrect; remove it.

This operation may be taken in two situations:

(A) a was an assumption or a guess, i.e. its BECAUSE property is NIL, and we want to remove that assumption.

(B) Some ambject on the BECAUSE property has had its BECAUSE property revoked.

In the former case, the actions to take are obvious. In the latter case, there may still be some other reason for keeping a true. If it exists, this alternative reason should be on the DNF property. Therefore, the following steps must be taken:

(2Ba) Remove the BECAUSE property from the association-list a.

(2Bb) Try to find some member rrrq of the DNF property of a, each of whose members has a BECAUSE property. If we find rrrq, put it on a as a BECAUSE property, and return. Otherwise, consider each member a' of the HELPSIMPLY property of a, and start again from step (2Ba) on it.

In the LISP program for ambeval, steps (2Ba)-(2Bb) are performed with "revokehypoth (a)" for case (A) and with "recheckfalse (a)" for case (B).

3.3 How to guess, and how to revoke a guess.

A "probble inference" may be on the type "for each t ⊷ u, if p(t) and q(t) hold, then it is a reasonable guess that r(t) occurs". The incertitude in this statement may arise in several ways:

(A) We know that the full and precise statement should be

let f be a u in

$$p(t) \wedge q(t) \wedge \neg v(t) \supset r(t) \;,$$

where v(t) is a relation which indicates some exceptional circumstance (like "the ceiling falls down") that we can usually ignore.

(B) There are not one but many exceptional circumstances, and they can not all be enumerated.

This can be reduced to case (A) in the following manner. We introduce a new predicate w and let w(t) mean "some of the exceptional circumstances in this rule holds". The rule is therefore

let t be a u in $p(t) \wedge q(t) \wedge \neg w(t) \supset r(t)$.

To express "v(t) is one of those exceptional circumstances", we then write

let t be a u in $v(t) \supset w(t)$.

(C) The implication let t be a u in $p(t) \wedge q(t) \supset r(t)$ has been obtained empirically, and we do not yet feel sure that it holds. This means that we want to assume or guess that the implication holds, and to perform inference from it, but we also want to be able to take back the assumption later if contrary evidence should appear. Crudely speaking, in cases (a) and (b) the true implication says that a guess can be made, where as in case (c) we guess that an implication holds. We shall now demostrate how these guesses can be made and revoked with ambeval.

Case A. We introduce a Boolean function unless, which is to be used instead of not in cases where the argument is usually false. Thus the statement shall go

let f be a u in p(t) ∧ q(t) ∧ unless(v(t)) ⊃r(t)

Ambeval needs the following operators to handle the function unless:

(1) An operator which checks t in each expression unless(t) and, if t does not have any BECAUSE property, gives unless(t) the property NIL under the attribute BECAUSE (which means that unless (t) is considered true with no reason);

(2) An operator which operates on recogn(exprizrrue). For each true ambject t, it checks wether unless(t) is in the system, and if it is, makes sure that it does not have any BECAUSE property.

The first of these two operators can immideatily be written down (all functions have already been introduced):

```
let ii be an unless in evalq(
        if assoc (BECAUSE, cdr( argument (1, xpr(ii)))) then NIL else
        putreason ( NIL, ambofxpr(ii))     )
```

The latter operator can be written

```
let ii be a recogn (EXPRIZTRUE) in evalq (revokehypoth  ( ambofxpr (
    select ( / λ (v) eq ( argument (1, xpr (car(v))),
                         ii  )/,
              cdr(assoc(CASES, cdr(unless))) ))))  (*)
```

We assume that the expression select (p,l) selects from the list l, the first member r that satisfies p(r). We write "xpr(car(v))" rather than "xpr(v)" because car(v) is an ambject (the case of the ambject unless) and cdr(v) is NIL, i.e. the list of reasons for car(v) ↚ unless.

Case B (indefinite number of "exeptional cases" or "unless" expressions). It has already been demonstrated that this can be reduced to case A.

Notice that as we have (by specialization)

$$p(a) \wedge q(a) \wedge \text{ unless } (w(a)) \supset r(a)$$

$$v(a) \supset w(a) \;,$$

and p(a) and q(a), but not v(a) are true, then unless(w(a)) will tentatively have the BECAUSE property NIL i.e. be true without reasons, and r(a) will be true. If we later set v(a) true, our operators above guarantee that r(a) is set false, unless it has some alternative reason.

(*) This use of the function recogn is incorrect under the conventions of section 2.6 but can be made to work.

Case C. The whole implication is a guess. We expressit by

"let t be a u in p(t) $\wedge$ q(t) $\supset$ r(t)"

or a logically equivalent expression. When evaluated by ambeval, this expression translates into a ro-expression and obtains an ambject e. Besides, the BECAUSE property of e is set NIL to indicate that e is an assumption. If the system knows for some tt that *tt and tt $\leftarrow\!\!\urcorner$ u   then it will evaluate "p(tt) $\wedge$ q(tt) $\supset$ r(tt)" and give the corresponding ambject ee the BECAUSEproperty (e), -- not NIL! -- unless it had previously a BECAUSE property. Also ee will be put on the HELPSIMPLY list of e. If the guess of the rule in e should later be taken back, the truth of ee will therefore also be re-inspected. The same is of course done to all conslusions that may have been drawn from ee.

3.4 Circularities

When guesses are revoked, circular arguments will easily occur. The purpose of the present section is to explain this phenomenon.

Let us introduce one more surface function, the prefix ↑ . It is used when we want to take back an assumption, and annihilates a previous !. If we do (↑ . EXPRESSION), where EXPRESSION has the property (BECAUSE), removehypoth(expression) is performed.

Suppose we do

(! AND LOG1 LOG2)

where LOG1 and LOG2 have not been used before. The following things will happen (although not necessarily in this order):

(1) The ambjects LOG1, LOG2, (AND LOG1 LOG2), and (' AND LOG1 LOG2) are created.

(2) An operator on the ambject AND puts ((LOG1 LOG2)) as the DNF property of (AND LOG1 LOG2); checks if both LOG1 and LOG2 have any BECAUSE property (because if they had, (LOG1 LOG2) would be put as the BECAUSE property of (AND LOG1 LOG2); fails; and returns.

(3) By the ! (AND LOG1 LOG2) is given NIL as one possible reason. Its DNF property is changed into (NIL (LOG1 LOG2)), and it obtains the BECAUSE property NIL. The ambject (' AND LOG1 LOG2) becomes at least implicitly a case of (RECOGN EXPRIZTRUE).

(4) By an operator on (SLASH AND (RECOGN EXPRIZTRUE)), the ambjects LOG1 and LOG2 are made true. Their DNF property is (((AND LOG1 LOG2))), i.e. a list of a list of the ambject (AND LOG1 LOG2). Their BECAUSE property is ((AND LOG1 LOG2)).

Now suppose we do later on:

```
( ↑ AND LOG1 LOG2)
```

OK

i.e. revoke the assumption. The following will happen:

(5) (AND LOG1 LOG2) loses its BECAUSE property, and the DNF property loses its NIL. If it was (NIL (LOG1 LOG2)), it goes into ((LOG1 LOG2)) .

(6) By the routine in section 3.1, we check the DNF property of (AND LOG1 LOG2) for some alternative reason. As both LOG1 and LOG2 now have a BECAUSE property, they are accepted as reasons, and the new BECAUSE property of (AND LOG1 LOG2) is (LOG1 LOG2).

The system now believes that LOG1 is true because (AND LOG1 LOG2) is true, and similarly for LOG2, and it believes that (AND LOG1 LOG2) is true because both LOG1 and LOG2 are true.

There are several ways of getting around this difficulty:

(a) When an assumption is revoked, we first remove the BECAUSE property from all its dirct and indirect consequences, and then run through those consequences again and look for alternative reasons in the DNF properties.

(b) We keep the revocation method essentially as we first designed it, but when we look through the DNF property of an ambject aaa for an alternative reason, we chain backwards through the BECAUSE properties and verify that the proposed BECAUSE property of aaa does not rely on aaa.

(c) We keep the revocation method as first designed, and accept that circularities occur. However, we have an operator which runs around in the data structure and looks for circularties. When it finds one, it removes the BECAUSE property of all members of the circle, and then tries to find an alternative reason in the DNF property of each of them.

In a certain sense, alternative (c) is less correct than the other ones. It admits that circularities can occur in the data base, and therefore, it admits incorrect conclusions. However, it is attractive for the following reasons:

(c1) The search for circularities can occur when the computer "is asleep", when it does not have anything else to do. In a time-sharing system, it can do this on low priority when the user of the incremental computer is doing his own thinking for a couple of minutes, but the program remains resident.

(c2) Search for circularities can be directed to important spots. Suppose the user has asked the computer "is it really true that ... (expression) ?"; that he posed the question with an indication of doubt, and that the computer should therefore think before it answers.

A reasonable reaction from the computer would then be to go rather far back through the BECAUSE tree, looking for circularities. If it does not find any, but does not either reach the end of the tree, the computer should (A) answer "yes, I belive so" (which it really does), and (B) insert on the queue of planned operations, a reminder to do a very-low-priority, very-time-consuming analysis of the BECAUSE tree for the given expression later on. On the other hand, if no member of a circular argument in the data structure is ever questioned or even used, then a system using alternative (c) will not waste time on eliminating this circularity.

(c3) Alternative (c) seems rather similar to the way humans function. With alternative (c), the really big circularities (the ones that we can not, and perhaps should not eliminate) will stay, whereas smaller circularities are removed whenever they are discovered. It would (probably) be impossible to analyse the entire belief structure of a human. Similarly, it may be practically almost impossible to analyse the complete data base of our incremental computer, if it has grown to the size that is required for handling practical problems. In both cases, the best we can do is to state the principle that circularities are incorrect, and remove them whenever we find them.

3.5 Conclusion

In this report, we have tried to outline the principles of the ambeval program. Many details are missing, and other details have been given a simplified treatment to facilitate the exposition. The reader is referred to a listing of the actual ambeval program for all details. Such a listing was issued as an appendix to this report.

Let us put down again the characteristics of ambeval programming: The program is written as a set of operators. These communicate only through facts, not through subroutine calls. In other words, one operator deposists a facts in the data base, and another operator is triggered by this fact to deposit some other fact, or to perform an action.

An operator which is triggered by the deposit of a fact, and which itself deposits a new fact, defacto performs an inference. The line of reasoning is stored in the ambeval system This makes it possible to guess and to revoke guesses.

Some examples of operators have appeared in this report. More examples will become available when work on the use of ambeval for actual problems has been completed.

The present LISP implementation of ambeval is very slow. It seems quite feasible and very desirable to integrate ambeval with eval (i.e. the LISP system itself) and implement the resulting extended LISP in machine code throughout.

References

1. Bertram Raphael

   SIR, a Computer Program for Semantic Information Retrieval

   PhD Thesis, MIT, Math Dept., Cambridge, Mass.

2. John McCarthy

   A Basis for a Mathematical Theory of Computation

   in [5].

3.* P.J. Landin

   A Correspondence Between Algol 60 and Church's

   Lambda-notation. Part I

   Comm. ACM 8 (1965), p. 89

4. L.A. Lombardi and Bertram Raphael

   LISP as the Language for an Incremental Computer

   in [6].

5. P. Braffort and D. Hirschberg, editors

   Computer Programming and Formal Systems

   North-Holland, 1963

6. Edmund C. Berkeley and Daniel G. Bobrow, editors

   The Programming Language LISP: Its Operation and Applications

   MIT Press, 1966

---

*) This reference is used on page 11. The reference [1,3] on page 1 should read [1].

Appendix 1: Truth-values for logical connectives in the ambiguity log

Truth-values: t = true, f = false, s = sometimes.

a	b	a∧b	a∨b	a⊃b	a?b	a/b	a≡b	a⊐b	a←
t	t	t	t	t	t	t	t	t	t
t	s	s	t	s	s	t	f	f	t
t	f	f	t	f	s	Я	f	f	f
t	Я	Я	Я	Я	t	Я	f	f	f
s	t	s	t	t	s	t	f	t	f
s	s	s	s	s	s	s	t	t	t
s	f	f	s	s	s	f	f	f	f
s	Я	Я	Я	Я	s	Я	f	f	f
f	t	f	t	t	s	Я	f	t	f
f	s	f	s	t	s	f	f	t	t
f	f	f	f	t	f	f	t	t	t
f	Я	Я	Я	Я	f	Я	f	f	f
Я	t	Я	Я	Я	t	Я	f	f	t
Я	s	Я	Я	Я	s	Я	f	f	t
Я	f	Я	Я	Я	f	Я	f	f	t
Я	Я	Я	Я	Я	Я	Я	t	t	t

a	a	*a
t	f	t
s	s	f
f	t	t
Я	Я	f