UPPSALA UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCES
NOTE NR 2
OCTOBER, 1967

The GPS expressed in LISP A

by

Erik Sandewall

1. Introduction and references.

This note gives one example of LISP A programming. We show a four-line program for Newell, Shaw and Simon's General Problems Solver, as described in

A. Newell, J.C. Shaw and H. Simon
Empirical Explorations with the Logic Theory Machine
in (a) Proc. Western Joint Comp. Conf., 1957,
and (b) E.A.:Feigenbaum and J. Feldman
Computers and Thought
McGraw-Hill Book Co, 1963.

LISP A is a LISP-like programming language for an incremental computer. A preliminary version was described in

Ambiguity Logic as a Basis for an Incremental Computer
Uppsala C.S. department, report ≠ 9.

We shall use here a slightly different LISP A, described in

Outline of LISP A
Uppsala C.S. department, note ≠ 1.

Note ≠ 1 describes only the changes, and assumes chapters 1 and 2 of report ≠ 9 as known.

2. The General Problem Solver

The GPS model (as described in "Empirical Explorations...") makes use of the following entities:

Objects, e.g. expressions in elementary propositional calculus. An object α conforms to an object β if α is obtained from β by substitution of subexpressions for variables.

Operators on objects e.g. the operator which transforms the object $A \supset B$ into the object $\sim B \supset \sim A$ (where A and B are arbitrary sub-expressions);

Differences between objects. Usually, a difference is expressed as a description of what has to be done to make one object conform to the other. Examples of difference descriptions are

"Add terms"
"Change grouping".

Goals, which may be of three kinds:

"Transform object A into object B",
"Reduce difference D between object A and object B",
"Apply operator Q to object A".

It might be argued that what we have here called differences should in fact be called goals. In that case, what we have called goals is in fact "meta-goals". Such distinctions are not of interest.

The GPS essentially consists of three rules, which tell how these three types of goals shall be processed. The rules are recursive, i.e. in order to accomplish some goal, we set up some sub-goals, which may set up new subgoals, possibly of the same type as the first goal, etc. As GPS documantion is easily available, we shall not reiterate these rules.

3. The LISP A program for the GPS

Our program assumes that there should be one unambigous ambject for each object, each operator (of which there are twelve, named r1, r2, . . r12 in the particular example discussed in "Empirical Explorations...), and each difference (of which there are seven in the same example). Also, if a, b, d, and q are unambigous objects, the goals

transform(a,b)
reducediff(a,b)
applyop(q,a)

(with the obvious meanings of these expressions) should be represented by unambigous ambjects. In general, however, the arguments of these three functions are ambigous, and then the value is an ambigous ambject.

With each operator ri there is associated a symbolic function $\overline{ri}$ which we also write bar(ri).

Simple example of representation. Suppose the operator $r4$ transforms the object a to the object b. If we evaluate the expression

transform(a,b),

the following ambjects will be introduced:

1. The ambject A. It has A as its ENAME property. The description of the object (which in the propositionalcalculus example is a formula, e.g. "$q \wedge q \supset r \vee s$") is a property under an attribute DESCR.

2. The ambject P. Similar to A.

3. An ambject $(\overline{R4}\ A)$. This stands for "the object obtained by applying operator $r4$ on object a". Its MEAN property is of course $(R4\ A)$. Like A , it has a DESCR property, which shows a after $r4$ has operated upon it. This DESCR property was assigned by a LISP A operator (to be distinguished from a GPS operator!) let E be an $r4$ in ...

which works because (' $\overline{R4}$ A) is a case of $\overline{R4}$. (Notice, by the way, that $\underline{R4}$ has a cases all the expressions that use $\overline{R4}$ as leading function, and is therefore ambigous, whereas $\underline{R4}$ (without bar) is unambigous).

4. An ambject (transform A B), whichobtains on its CASES property, ambjects like ($\overline{R4}$ A). In general, all ambjects ($\overline{R1}$ ($\overline{R2}$ (... ($\overline{Rk}$ A) ...))) which are obtained from A by successive transformations, and which conform to B (by comparison of DESCR properties) are obtained as CASES of (transform A B) , if the LISP A system is permitted to run long enough. - "transform" stands for the value of the atom TRANSFORM.

The functions for the three goals, i.e. "transform", "reducediff", and "applyop" are defined as rho-expressions, i.e. the value of each of those atoms is a rho-expression. On M-language form, the function definitions go:

```
transform(a,b) = if  null(finddiff(a,b))  then  a  else
                     transform( reducediff(a,b), b)
reducediff(a,b) = applyop( signifop(finddiff(a,b)), a)
applyop(q,a)    = bar(q)  (transform(a,domain(q)) )        (*)
```

This is the entire GPS program. The purpose of the functions is as follows: transform (a,b) transforms the object a into the object b. It does this by first transforming a into some object a' which is (in some sense) closer to b, and then trying transform(a',b).

reducediff(a,b) reduces the difference between a and b. It returns some object a' which has manifestly been obtained from a through the application of permitted operators, and which is closer to b than a was.

(*)
The meaning of this expression is that we first evaluate the value $\overline{q}$ of bar(q) and then consider it as a function which is evaluated with the value of transform(a, domain(q)) as argument. This is possible in LISP A. The corresponding S-expression goes

```
applyop  =  (RHO  (Q A)
                  ( (BAR Q)
                    (TRANSFORM  A  (DOMAIN Q))  ))
```

applyop(q,a) applies operator q to object a by first transforming to a form which is in the domain of q .

The above definition makes use of some auxiliary functions, which have to be written specifically for each application. They are:

finddiff(a,b), an eta-expression which takes unambigous ambjects (for objects or object classes) arguments, and returns

NIL if a and b are considered equal according to their DESCR property;

an unambigous difference description otherwise.

signifop(d), an eta-expression which takes a difference description as argument and returns an operator that seems significant to the difference, as value. (In the example, this function is merely an encoding of the "connection table" on the bottom of page 285 in Computers and Thought).

domain(q), an eta-expression which takes an operator q as argument and returns the domain of q, i.e. the class of objects on which q is defined. For example, if q is the operator that transforms $A \supset B$ into $\sim B \supset \sim A$, then domain(q) is the class of formulae on the for $A \supset B$.

bar(q) an eta-expression which takes an operator q as argument and returns the corresponding symbolic function $\overline{q}$ as value. (*)

(*) The reason we can not identify q and $\overline{q}$ is that $\overline{q}$ must take a lot of quoted expressions as cases and therefore be unambigous,whereas q must be an unambigous ambject for each operator (because the definition of applyop as a rhoexpression relies on the idea that the first argument is an ambiguity between operators, e.g. r2? r5? r6, and each unambigous operator shall be used individually for q).

This is a consequence of the connection table, which says (in the example)

```
signifop(d) = if  d = add-terms
                then  r3 ? r7 ? r9 ? r10 ? r11 ? r12  else
              if  d = delete-terms
                then  r3 ? r7 ? r8 ? r11 ? r12  else ...

              if  d = change-position then  r1 ? r2.
```

Because it is a rho-expression, when applyop is given an ambigous first argument, it will let q'(transform(a,domain(q'))) be evaluated for each unambigous case q' of its first argument. For each such case, transform is called, so reducediff is called, so signifop is called, so new branching occurs.

The above recursive definition could therefore never be used in ordinary LISP, because it would lead to a depth-first search through all possible transformations. In LISP A, branches are studied and continued according to priority evaluation, so the study of a branch may be given up and the study of any other branch taken up instead. Study of the first branch may later be resumed.

4. How the rho-functions work.

The eta-functions finddiff, signifop, domain and bar work just like functions in ordinary LISP, but the rho-functions transform, reducediff, and applyop have to be explained slightly more in detail. Consider the evaluation of transform(a,b) on some level in the middle of recursion. Both arguments, a and b, are ambjects, and the first argument is most likely ambigous. It carries a list of CASES, some of which may be unambigous. The LISP A evaluator now introduces a new ambject tab, whose MEAN property ("meaning") is

list(transform, a,b).

The ambject tab will be the value from evaluating transform(a,b), but before returning tab, the function evaluator evaluates the expression

```
if null(finddiff(aa,b)) then aa else
transform( reducediff(aa,b), b)
```

for unambigous aa that are cases of the first argument a. The value of such an expression is a new ambject, as transform returns ambjects as values. It is set a CASE of the ambjects tab. The CASE relation is transitive, so an ambject obtained when recursion terminates (i.e. according to the clause "if null(finddiff(a,b)) then a") is not merely a CASE of the ambject transform(a,b) that immideatily evaluated it, but also a CASE of transform(a,b) on the outermost level of recursion.

We stated that "before returning tab, the function evaluator evaluates the expression ... and sets the value as a CASE of tab". This is not altogether true, because it depends on priority. At least in some branches, the evaluation of this expression may have been postponed. If search in high-priority branches was unsuccessful, the function evaluator may return to such postponed branches (i.e. reach them in the priority queue). More CASES will then be added to the ambject a that was an argument to "our" expression transform(a,b) above. However, the specifications of how rho-expressions are to be handled guarantees that such new cases will be followed up (upwards in recursion) just like if they had been incorporated as CASES before tab was returned.

The other two rho-functions are handled analogously.

Through such mechanisms, the LISP A system can evaluate an expression transform(a,b); find various transformations of a into b, and set each transformed a as a CASE of the ambject transform(a,b). Besides the functions specified above, there must of course be functions for priority evaluation, which let the most promising branches be evaluated first and cut branches that are too unpromising. Also, if the GPS task is a job in itself (rather than a subroutine in some larger job) we will usually not evaluate transform(a,b), but instead describe(transform(a,b)), where describe is an eta-expression which goes into the CASES property of its argument and prints out the MEAN property, so that the sequence of operators that were used for the transformation is obtained. By the notation in report ≠ 9, we have

describe (t) = usenames(cdr (assoc(CASES, cdr(t))))

4. How the function applyop and the symbolic functions (the GPS operators) work.

The definition of applyop was

applyop(q,a) = bar(q) transform(a,domain(q))) .

This definition shall be studied more in detail. Let us first make the convention that all $\overline{q}$ shall belong to the class of "strongly regular" symbolic functions, characterized by

$$*_t \leftrightarrow *_{\overline{q}(t)}$$

and $\overline{q}(¶) = ¶.$

Some conclusions from the first of these axioms are executed through

(1) let e be a strongly-regular / unambigous in

isstar(argument(1,e))

and

(2) let e be a strongly-regular / unambargument(1) in

isstar(ambofxpr(e))

where unambigous is the ambject which has all unambigous expressions as cases on their quoted form, and unambargument(1) is the ambject which has all expressions whose first argument is unambigous, as cases on their quoted form.

Consider now the evaluation of

applyop(q,a)

on some level in recursion. q was obtained as the value of signifop(---), and is most likely ambigous. a can be proved to be unambigous. Applyop is defined as a rho-expression. For each case qi of q, it therefore evaluates the ambject TAQ = transform(a,domain(qi)), which is in the general case ambigous, and then evaluates $\overline{qi}$(taq). This introduces a new ambject with the MEAN property ($\overline{QI}$ TAQ). For each qi, this ambject becomes a case of applyop(q,a), but by the first axiom above, it can not be unambigous.

However, in order to continue recursion, we are interested in unambigous cases of applyop(q,a).

This is arranged as follows. The LISP-A-operator (2) above is re-written on the form

(3) let e be a strongly-regular in
let a be an argument(1,e) in
isstar(argument(0,e)(a))

The meaning of the last line is that argument(0,e) (i.e. the function symbol) should be used as a function with the argument a. In S-notation this is more clear:

(ISSTAR ((ARGUMENT 0 E) A))

We are again making use of the LISP A innovation that we can evaluate a function and immideatily use it.

In the example, suppose taq has the unambgious case taqj, i.e. we have *taqj and taqj ↤ taq. Clearly, we want $\overline{qi}$(taqj), which is unambigous. We evaluate $\overline{qi}$(taq) as argued above. This causes the ambject (' $\overline{QI}$ TAQ) to be set as a CASE of $\overline{QI}$, which in its turn a case of strongly-regular. Therefore, the above operator is triggered with e = (' $\overline{QI}$ TAQ), which is unambigous. Then argument(1,e) is taq, and the inner let-expression is applied with e.g. a = taqj. On the third line, argument(0,e) evaluates into $\overline{QI}$ so the ambject

(ISSTAR ($\overline{QI}$ TAQJ))

is evaluated and set as a case of the above LISP-A-operator. This causes it to be declared true, so the ambject ($\overline{QI}$ TAQJ) is established as unambigous. By elementary axious, it is a CASE of ($\overline{QI}$ TAQ) and therefore of applyop[q,a] . This is the ambject we need, the ambject which can be carried upwards in recursion.

Notice that the above operator (3) is applicable to all strongly symbolic functions, not merely to the symbolic function that serve as GPS operators.

5. How to make applyop more efficient.

We have

applyop(q,a) = bar(q)(transform(a,domain(q))).

If some case qi of q can not possibly be applied to a,
transform(a,domain(qi))
will never be assigned any cases, and therefore operator (3) in the preceeding section never comes through (because there never comes any a for "let a be a argument(1,e)"). However, there will be an ambject qi(taq) (with taq as in last section) in memory, and this takes space and extra work.

Sometimes, it is possible to say before the evaluation of
transform(a,domain(qi))
has even started, that it will not succeed. In Empirical Explorations...", a "preliminary test of feasibility" is performed to prune dead branches as quickly as possible. A corresponding test may be introduced into our formulation of the GPS by changing the definition of applyop into the rho-expression given by

applyop(q,a) = if preltest(q,a) then ¶ else
bar(q) (transform(a,domain(q)))

where preltest(q,a) returns the value T if the (unambigous) operator q can not successfully be applied to the object a. (Notice that ¶ ↜ t is a trivial statement for any ambject).

6. Conclusions.

The program for the General Problem Solver becomes very simple and compact if LISP A facilities can be used. The program for the abstract GPS takes four lines. Specifications for additional functions, which are needed for the application of the GPS to some problem environment (e.g. the functions finddiff and domain) are simple and straight-forward. We have not "swept the problems under the carpet".

LISP A is an extremely slow programming system. The LISP A version of GPS can therefore never be practicially useful. However, the ease of programming it offers makes it feasible to write down and check out more complicated Problem Solvers, where more provisions are taken.

The LISP A system is essentially taking care of the tree search aspects of GPS. By generalization, we can therefore conjecture that LISP A shall prove a useful tool for quick programming of other search processes as well.

The LISP A system (i.e. the function evala) does not "contain" any built-in GPS. In fact, it was not at all designed with GPS in mind, but intended for automatic inference("incremental computer"). Distinction: the incremental computer works forward blindly in a stimulus-response manner. Its responses are determined by priorities, but these are not necessarily assigned with some goal in mind. (If there is to be any goal direction, it has to be built into the priority evaluation functions). The GPS does work toward explicit goals.

The conclusion that goal-directed behavior can be simulated by blind stimulus-response processes on a higher level than the computer program, is noteworthy in itself.