TR-015

Logic Programming
— Past, Present and Future —

by

J.A. Robinson
(Syracuse University, U.S.A.)

June, 1983

LOGIC PROGRAMMING

- PAST, PRESENT AND FUTURE -

J. A. Robinson

Syracuse University

[The following text is an edited and condensed transcript of an ICOT Public Lecture given in Tokyo on 10 February 1983]

Thank you very much. I wish I could speak to you in Japanese. Next time perhaps I will be able to do so with the help of a Fifth Generation computer.

In thinking about the history and future of the idea of logic programming it helps to distinguish the following periods:

distant past	(1879 - 1970)
near past	(1971 - 1980)
present	(1981 - 1990)
near future	(1991 - 2000)
distant future	(2001 - ?)

1.0 THE DISTANT PAST (1879 - 1970)

The distant past starts with an important milestone in the history of ideas. 1879 happens to be the year in which Albert Einstein was born. But in that year something else was born, namely, the predicate calculus, as we now have it. It was invented by one man, Gottlob Frege, a mathematician whose goal was to analyze completely the formal structure of pure thought. Frege called his system the Begriffschrifft, a word he appears to have also invented . It seems to mean something like "notation for concepts". He thought of it as a universal language in which every possible form of rational thought that could enter into a piece of deductive reasoning could be represented in a systematic and mathematically precise way.

And so it proved. I think that the history of this systematic notation of Frege's has borne out his faith in it. The rest of the distant past is essentially about the development in one particular line of that notation. The particular line is what we might call "computational predicate calculus", the line that is always seeking algorithms in which the processes of deduction are captured in a systematic way. As Frege then saw it, these processes were to be mathematically precisely presented, so that they could be studied formally. However, he was not directly concerned with computational issues as such.

So the distant past of logic programming is the history of computational logic. There was a long period, of about thirty five years, after the introduction of the Begriffschrifft before anything really significant happened. It's fashionable to mention in the history of the predicate calculus the famous work of Whitehead and Russell - Principia Mathematica. However, that work is not really part of computational logic, but part of another branch of the development in which Frege was also very interested, namely, the effort to analyze to their very foundations the basic ideas of mathematics: the notions of function, infinity, set, and so on. Frege believed, and so did Whitehead and Russell, that these notions could be analyzed in purely logical terms, and that is what that particular line of development - logicism - was endeavouring to do.

In computational logic the first really significant work after Frege was that of Lowenheim in 1915. This began a fruitful period of exploration and discovery culminating in about 1930. At about that time, the discovery of what we now think of as the fundamental theorem of the predicate calculus was made independently by the French algebraist and logician, Jacques Herbrand, who was writing his Ph.D. thesis in 1929 and in 1930, 21 years old at that time; a Norwegian, Thoralf Skolem, a mature professional mathematician working throughout the 1920's on the same problem; and Kurt Godel, an Austrian mathematician; all of whom were attempting to prove about the predicate calculus the basic fact which Frege took on faith when he invented it in the first place, namely, that it is indeed a complete system of notation, that it actually does do everything that it is intended to do.

The intention behind the language is that it should provide a formal proof of every sentence in the language which is logically valid - that is, true under all possible interpretations - and that this proof should be systematically constructible, given the sentence. Godel, Herbrand and Skolem showed in different ways that this is the case. It is to these men that we owe today's predicate calculus proof procedures.

Herbrand gave several versions of the proof procedure, one of which involved him in the idea we now call "unification".

Of course in 1930 there were no computers to run the procedure on, and indeed computers did not arrive on the scene until the early 1950's. So nobody was able to think of programming the proof procedure for a modern computer until about 1955 when a Dutchman, Evert Beth, decided to try it, and he was followed by others we should remember, Stig Kanger and Dag Prawitz from Sweden, Paul Gilmore, Hao Wang, Martin Davis and Hilary Putnam from the USA, -- all of them decided to try the now 25-year-old process on the computers of that era.

In doing so, Dag Prawitz, in 1960, revived the unification notion, and used it in one of his early programs.

It was the experience of these investigators that the algorithm, as it was formulated in 1930, wasn't very well thought out from the modern computational complexity point of view, and that it involved enormous combinatorial explosions. They were somewhat disappointed in its performance.

At this stage I myself became involved in this effort. About 1961 I started to study these papers, and it occurred to me that there were a few tricks one might use to improve the computational performance of the basic algorithm. In the course of that work, I stumbled across the idea that's now known as "resolution", which was a way of involving the unification concept right at the very heart of the proof system that one was dealing with.

I will say more about the developments that ensued after the early time when resolution began to be the way people attempted to make the computer efficiently carry out the basic process.

But I would first like to mention another thread in this brief history of logic programming. This was not quite as direct a part of the development of logic proper, but certainly it belongs in the logic programming story. In the middle 1960's two gentlemen whose names are not as widely known as they should be - Michael Foster and Ted Elcock - were experimenting with a programming formalism that they called ABSYS (for ABerdeen SYStem). Later they renamed it ABSET (for ABerdeen SET language). It was a computing language based on the idea that the programmer would simply make assertions.

These assertions, or sentences, would be, so to speak, axioms that the programmer believed to be true. They would be entered into the memory of the computer, and then used as the premisses of deductive inferences when a query, as we would now call it, was submitted to the system. These assertions would then be invoked automatically in deducing the answer to the query.

This was very interesting early attempt to do what we now think of as logic programming. It was not, however, carried through in the context of the formal predicate calculus, but was done intuitively in more or less ordinary mathematical notation. It was a rather complex system, and didn't behave quite as efficiently as one could have wished. But anyone who wants to go back to the beginnings of logic programming should be aware of that work.

Well, resolution spread around and many people took it up, including an early pioneer in resolution logic programming, Cordell Greene, in Stanford in the late 60's, who attempted to use resolution in essentially the modern way as the basis of a logic programming system. He thought of it as a question answering system. Rightly enough; that's what logic programming really is all about. His work attracted quite a lot of attention;--some good, some bad. The good part was that at last here was a systematic plan for a question-answering computing system. It seemed to be very general and potentially applicable to a wide range of problems. The bad part was that the particular resolution algorithm underlying it was still computationally complex enough to limit the applications of the system to fairly small problems. On larger problems, Greene's system would again run into combinatorial explosions. This was once again disappointing.

So at the end of the 1960's and at the beginning of the 1970's there were some rather negative comments made, especially from the direction of Cambridge, Massachusetts, to the effect that doing artificial intelligence computing by <u>logic</u>, and especially by <u>resolution</u>, was an extremely silly thing to do; and that the proper way to proceed was in a different manner. The MIT system PLANNER emerged during that controversy. It was another line of development that I think now is joining again into the main stream.

A number of people - Robert Kowalski, Donald Kuehner, David Luckham, Donald Loveland, and others - were attacking relentlessly the problem of the combinatorial complexity of resolution:--and a lot of ideas were tried. The winning one turned out to be to restrict the resolution rule so that the deductive structures that would be generated by the algorithm would be <u>linear</u> in form.

That would mean that each proof would be a tree structure with one main branch. Every inferred clause would lie on the main branch and would resolve the previous clause on the main branch with one of the input clauses.

Another line was to take the unification process, and try to make it faster, and improve its performance.

Several people - I was one - were working on this: Bob Boyer and J Moore joined us at that time. This was in Edinburgh in Scotland. They thought of all sorts of wonderful algorithmic tricks including structure sharing for speeding up the resolution process. Essentially, the development down to here can be summed up by saying: all the pieces were now available for PROLOG.

2.0 THE NEAR PAST (1971 - 1980)

It took someone like Alain Colmerauer to see all those pieces and put them together into a homogeneous system. First of all, Colmerauer invented one that he called SYSTEM Q , and then he named it (or it was his colleague Philippe Roussel who named it, or it may have even been Roussel's wife who named it) PROLOG; we can't quite find out who thought of that name.

At any rate PROLOG was born in Marseille in 1971. Put simply, it consisted of a linear resolution system in which the clauses involved in the problem were restricted to be Horn-clauses, together with an interpretation which is due to Kowalski of what is happening when you run the system - an interpretation which directly transforms the theorem proving process into a more traditional computation process. Each step is the invocation of a procedure which then returns some kind of result to its caller. All of those computational notions were exploited by Kowalski in this procedural interpretation of linear Horn-clause resolution systems. This then was how logic programming as a concept came about.

It was Kowalski who saw all this in PROLOG. He saw how to look at it in both ways: first as logic; second as computing.

The near past, then, begins with PROLOG springing fully grown from the head of Colmerauer, and with Kowalski beginning his crusade as a tireless proponent of the idea, effectively spreading the news. PROLOG existed! People very quickly saw its virtues and began to use it. Kowalski's 1974 IFIP address was an extremely influential exposition of logic programming in general and PROLOG in particular. It was he who first sparked the rapid growth of interest in logic programming.

This caused a number of the best younger computer scientists in Europe to take up logic programming as their main activity. I would like just to mention the main ones that I know: Sten-Ake Tarnlund from Sweden; Keith Clark from England; Maarten van Emden from Holland; Maurice Bruynooghe from Belgium; Peter Szeredi from Hungary; Herve Gallaire from France; David Warren, our colleague here today; Luis and Fernando Pereira from Portugal: all these people collectively gave an enormous impetus to logic programming. It was quite remarkable what force gathered behind the idea due to these splendid people.

We also saw a rather quick organizational development; as illustrated, for example, by the excellent book Logic Programming edited by Clark and Tarnlund, which is a record of an international Workshop of the whole community of logic programming researchers, telling their ideas for developing and applying the notion. This I think helped a great deal. We have now had further Workshops in Syracuse (Spring 1981) and Los Angeles (Summer 1981); and one is planned for Summer 1983 in Portugal. In Summer 1982 we had the First International Logic Programming Conference in Marseille, and plan the Second for Summer 1984 in Uppsala, Sweden.

Logic programming has been helped by some excellent books: Kowalski's Logic for Problem Solving; the PROLOG manual of Clocksin and Mellish; and I just mentioned Clark and Tarnlund's book. But of course we've got to pay our respect to the wonderful implementation that PROLOG was given after the original Marseille FORTRAN-based implementation.

Logic programming has been extremely fortunate to have David Warren's fantastic Edinburgh DEC-10 PROLOG, which I think really pushed logic programming over the top and made it into a useful tool for all manner of purposes. One cannot praise highly enough David's influence through that implementation.

In 1973 Ernie Sibert and I decided to implement a logic programming system in LISP at Syracuse. We call our system LOGLISP. The general idea behind LOGLISP is to try to take the logic programming notion and to blend it as nicely as possible with the function programming notion exemplified by LISP. LISP is the most famous case, but now there are purer and more elegant function programming systems such as David Turner's SASL and KRC, Peter Henderson's LISPKIT, and the one spoken about by John Backus in his 1977 Turing Award lecture.

Function programming seems on the surface to be an independent and somewhat separate notion from logic programming.

My contention is that they are both examples of a more fundamental single common idea, which we might (remembering the Aberdeen idea) call "assertional programming" - a type of programming in which what you do is assert some sentences to be true, and then ask for others to be deduced as a consequence.

In logic programming, those asserted sentences happen to be conditionals. In function programming, they happen to be equations. But that's a really only a superficial difference. I think the main point to notice is that when we run systems of either kind we are running deductive engines; we are asking them to make deductions for us.

So LOGLISP, which we are currently finishing up at Syracuse, is an attempt to embody both styles of programming within one framework. Other people such as Jan Komorowski of Linkoping in Sweden - now at Harvard University - also have tried to combine LISP and logic programming and of course I needn't point out to this audience that this is very much a theme in your own Fifth Generation Project.

The beginning of the Fifth Generation Project is the great event which marks the end of the near past. Quite suddenly we in the West had this delightful surprise. We found that here in Japan you had been quietly studying this idea, unknown to us, and had spotted it for what it was, namely, a beautiful, strong technique which could be exploited in the ways that you saw. For us, this was a wonderful way to end the 1970's.

I think one should say, instead of adopting logic programming as a central idea in your notion, what you really are doing is adopting <u>assertional</u> <u>programming</u> as the central idea. Because I have heard over and over again in visiting research groups here that same idea, to combine logic programming with function programming.

So we end the near past with a fine orchestral climax: and we enter the present.

3.0 THE PRESENT (1981 - 1990)

Now, we can't discuss the present in this talk in historical style, since it is not yet over. Instead, I would like to offer some observations about where we are going and what we ought to try to do.

I think there are some trends that we should be anxious about. I am afraid that the very success of PROLOG, which has been so resounding, may have some unfortunate aspects. For example, I regret (for similar reasons to Dijkstra's concerning GOTO) that PROLOG has the CUT feature in it, and that PROLOG programmers are encouraged to be ingenious in managing the particular way in which PROLOG develops that basic tree construction. It happens to do it depth first by backtracking, visiting all the nodes in the tree.

That's not necessary feature of a logic programming system: it happens to be the PROLOG way. It would be better if the details of that were invisible to the user; not thrust upon the user as one of the main things the user should be clever about in writing programs.

So, CUT is not a good thing, but then it may already be on its way out since it is a serial notion. As more and more parallel PROLOG implementations come along, you won't be doing backtracking internally, you will be doing tree development in a holistic, parallel manner. The intuitions behind the PROLOG programmers' design of PROLOG algorithms will then change and move to a higher level. And that will be good.

What PROLOG is really after in the CUT construction is a way for the programmer to plan computational economies in the construction of that tree. And of course one would like the programmer to be able to pass along advice to the system about what parts of the tree to neglect as being unnecessary, given that developments in the computation have reached a certain state, which couldn't be detected until run time. It's quite desirable, it seems to me, to provide the programmer with some way of influencing the efficiency of the tree development. Not, however, at the expense of intelligibility of the program!

A somewhat related point is that it ought not to make any difference in what order we assert the components of a conjunction, because logically they have no particular order. To give them an order is to superpose something else on what you are saying.

Within a clause, we ought not to have to worry about the order. Nor should we have to worry about the order of the clauses among themselves.

In short, we ought not to incorporate into the logical notation itself particular conventions about how to manage the details of the deductive search. Such details as the processor cannot be expected to decide wisely must be managed by the programmer through control inputs. But these should be separate from the logical inputs.

We need to keep in mind that logic programming in general is not to be identified with PROLOG, in particular. The relationship is that PROLOG is an implementation, a particular realization of the logic programming notion. I would even say that more generally, logic programming is not totally to be identified with Horn clause resolution programming. That just happens again to be a special case and a very good one, as we have seen, of the general idea of deductive computing from assumptions.

You might even go further still, and say it's not even really limited to the first order predicate calculus. After all, there are other interesting logics ; there is higher order logic; various flavours of modal logic, and so on. There are all sorts of rich formalisms that it might be thinkable one day to use in the way we now use a restricted predicate calculus to do logic programming.

So, I think we should preserve the terminology, and keep logic programming as a separate concept, and then have individual notions for various special cases of it.

I think we ought, in the same spirit, to contrast the general idea of a logic programming system with that of a complete programming environment. It seems to me that some of the things that you have to do in the various PROLOGS I have met are strange. You have to, for example, make side-effects take place, like printing, by attempting to prove a sentence; and in the act of trying to prove it, somehow off to the side, events take place. That doesn't seem to be very good conceptually. I think it's better to be honest about imperative programming; if you want something to <u>happen</u>, you should, I think, have facilities available for saying so and for <u>making</u> them happen. Your assertional semantics won't then be all cluttered up with side-effects.

I think another point which should be made about PROLOG is that it overstresses the role played by relations in assertional programming. Relations have a very important role, of course, but they are not everything.

It sometimes seems to me that we have returned to the earliest days of computing, when in expressing the evaluation of an expression, one had to introduce names for intermediate values and store them in cells with those names; finally there would be a cell with one's answer in it. Of course, the intermediate naming of steps in a successive evaluation of an expression is something that we really don't want to have to do. And it seemed to be progress when FORTRAN arrived, and allowed one to just write the expression down, and have it evaluated without having in assembly language oneself to name all those intermediate results.

If you look at some PROLOG programs where deeply nested expression are involved, you suddenly find yourself back in those days, having to name intermediate stages of a successive nested evaluation in order to come out at the end with a value. I don't think that the expression itself is unlogical - it's a term, after all - and I would prefer to elevate functions to the same level as relations, as Frege did in the original design of the predicate calculus.

I know that it's literally true that a function is just a special kind of relation. But you can turn that around, and you can observe also with equal merit that a relation is just a special kind of function. As a matter of fact, that's how Frege saw it. For him, a relation is a function from tuples of things to truth values. And so, you think of evaluating a relation in just the same way as you think of evaluating any other function. It's just a different target domain.

Well, this provokes us to ask the question: What is the best total programming environment? What should be the elements of it? If we want to have it contain editors , I/O commands, and other kinds of side-effecting machinery, we had better think it all out carefully so as not to mess up one of our most magnificent tools, namely, the logic programming formalism. It's surely got to be part of that environment, but we don't want to overload it, it seems to me, with all these other duties as well.

I have more anxieties. Hitherto, all the logic programming systems that we have had experience with, have been small. What do I mean by that? Mainly that they have been running on machines in whose main memory all of the assumptions were stored and thus randomly accessible through good indexing and associative retrieval methods.

What happens when we go to larger systems, where we can't put it all in the main memory? We are going to have to work with essentially disk-based virtual memories. And so we have to face the problem of the slowing down of the accessing to the assumptions, which is a little worrisome. It isn't clear to me how, if we are going to get to very large systems, we are going to be able to get the speed-ups that the Fifth Generation Project is talking about.

Today's speeds in LIPS - logical inferences per second - of logic programming systems are in the order of ten to the four. If we are going by 1990 to get up to ten to the nine, we've got to think out where that speedup is going to come from. It seems to me that if we can stay inside the main memory of the machine, we can quite happily plan on that speed-up. By going to parallel working, we can probably gain a factor of a hundred. By going to 1990 hardware we probably get another factor of a hundred. The remaining factor of ten we can hope to get by being even cleverer than we have been so far in organizing the basic algorithms.

So, I think a ten to the five speed-up is reasonable, provided that we can do it in fast memory. But if we have to get our clauses from disk memory, we have a problem to do that fast. Yet, how else are we to store a terabyte of information?

I confess that I sometimes have a twinge of anxiety about your having made logic programming the central theme in your Fifth Generation Project. I wonder whether your great confidence in this idea is going to be justified. There are some risks involved, as you well know, in putting this idea in the center. It is really an experiment. I think it's overwhelmingly probable that the experiment is going to be successful. But there are some hazards. I will say more about these in a moment.

A final, general worry - what's going to happen to logic programming, as a pure abstract idea, when you people get through with it? Everybody now is paying intense attention to the paradigm; changing it, experimenting with it in various ways with different motives. Something is going to happen to it, and I have the anxiety that it might not always be for the best. We have to try to guide the development now in this decade that we are just beginning, so that at the end of the decade we have a notion of logic programming systems that we can be proud of, a notion that is still elegant, powerful, and simple, and indeed that has all the virtues that logic programming now seems to have as an idea.

Let's hope that in making logic programming into a practical success on a large scale we don't have to sacrifice any of that elegance and beauty. I sometimes feel a little nervous when I see papers and listen to discussions in which logic programming is being blended in with everything under the sun. That's perhaps an unnecessary worry. I hope so.

Lastly, since I love LISP very much, as do a lot of other people, I hope that LISP, which is a beautiful thing, doesn't disappear. I am not so fanatical a logic programming proponent as to want LISP to be defeated, and to be superceded entirely by something like PROLOG. As I said earlier, the proper line is for both of them to become what each of them is trying to be, namely, an assertional programming system. So, I want LISP to survive - not necessarily down to the smallest detail - but as the basic idea of a lambda calculus based formalism, with a universal data structure of the the dotted pair. That's a beautiful and powerful idea.

So, let's not destroy LISP in making logic programming a success.

We really do have a wonderful opportunity to do good work on the paradigm of logic programming. Consider what Peter Landin did in the early 60's with LISP. He set out to show what surprisingly enough McCarthy, in inventing LISP, hadn't realized fully, that LISP was essentially the lambda calculus. He explained this with a marvellously elegant abstract machine, the SECD machine. This work of Landin's was I think extremely important and very beautiful work.

If you look at the work of the modern function programming researchers, like David Turner and Peter Henderson, you find a similar hunger for elegance there, which I personally react to very positively. I think that it's important to go for elegance and beauty in these mathematical engineering quests. You can't really go far wrong if it's beautiful.

We don't want logic programming to spawn off kludges. That would be very distressing. One of the unfortunate themes in the last fifteen or twenty years in artificial intelligence programming has been the tendency to create effective but ugly software. Let's try to avoid that.

An example of an opportunity for greater elegance is unification. I believe, and so do many others, that unification is a very powerful idea, which can explain a number of other ideas that have arisen in computer science, very well, very simply, and properly. I think it's the underlying mechanism for all processes of parameter passing as between function calls and function activations.

That's of course how PROLOG sees it; how Kowalski's procedural interpretation sees it. But it's never really been tried, as far as I am aware, in the function programming context. In ALGOL, PASCAL, LISP and so on , the parameter passing corresponds to a one-sided matching of formal parameter with actual parameter. And the actual parameter in such a comparison doesn't change. Only the formal parameter changes. And it's set to be the same as the actual, and then the body of the procedure is executed.

We now have a chance to see what happens to the functional programming situation, when we generalize parameter passing through making it into a two-sided exchange of information.

Kenneth Kahn and Harvey Abramson have both looked into the design of function programming systems, in which unification is the leading principle.

Also, I think it's now clear to many people - certainly to Colmerauer and also to a number of people I have talked to here in Japan - that one can very readily compute with what otherwise might be described as infinite expressions. They are not really infinite; they are representations of infinite things. The representation is done by means of pointers which can introduce cycles into the structure. LISP has dealt with such structures for years, but furtively. The use of RPLACA and RPLACD was thought to be "not quite respectable" and in any case dangerous.

Unification can perfectly well be generalized, and now is in many systems, to handle expressions of that character also, as well as the more usual finite expressions that we originally had in mind.

If you do that, you get the ability to represent streams, and also to introduce lazy evaluation into deductive computing, and many other good things. This is being worked out by a number of people including many groups in Japan.

As your Fifth Generation Project plans point out, we now have a chance to develop new architectures, to incorporate various kinds of parallelism, and to go for very large database applications. I earlier alluded to the worry that I have, whether you can have both huge collections of clauses up in the terabyte range with a gigalips of speed. It seems to me that we've got a problem there that I personally don't know yet how to cope with. However, the new technology is beckoning.

I think we are ready now for something like a Knuth treatise on <u>logic programming methodology</u>. This would make a major impact on the world community of computer scientists who may not have heard of logic programming yet. Perhaps Sten-Ake Tarnlund and Keith Clark, or Ehud Shapiro, or Maarten van Emden, would be good people to do it. They should really make a definitive attempt to write out what it is that logic progamming has going for it. I know there are books already. I know Kowalski has an introduction to the ideas. That's not quite what I have in mind. I allude to Knuth, because everybody knows what a wonderful job he has done for, so to speak, von Neumann computing; and logic programming needs a Knuth now. Perhaps, Knuth himself - who knows? - may get interested.

Another thing that worries me is the identification of logic programming with artificial intelligence as a movement in the history of ideas. It seems to me that they aren't the same at all, and my advice is: we should try to keep logic programming well apart from artificial intelligence; not try to hook them together. For one thing, I believe that artificial intelligence is just about to go down into another of its periodic troughs. If you look at the history of AI, you will see that it's been rather up and down: excessive enthusiasm followed by equally excessive disappointment. When you begin to see lots of superficial journalism and lots of television interviews with well-known faces, you begin to think that the wrong forces are at work. A good scientific trend happens more quietly than that, and doesn't need the kind of media exposure that AI seems to be getting, if not actually to be seeking.

I think that many of the famous accomplishments in AI are benign kludges, that is to say, I don't think you can extract from them, successful as some of them are, any systematic deep fundamental science. It's not always clear why things work well, if they work well. It seems to me that AI has got a long way to go before it becomes anything like a science; before it deserves that label. It seems to me mostly to consist of very worthwhile aspirations. Lots of good undertakings are afoot, but to aspire is not the same thing as to achieve. You have to do the work as well as talk about doing it.

For example, I feel that some of the propaganda that the notion of "expert systems" is now getting in the press, is slightly misleading. If you look at the well-known examples, for example, at MYCIN, or PROSPECTOR, or MACSYMA (these are successful examples; don't get me wrong!) and if you ask why are they successful I think you will see that it isn't the methodology that was followed out in constructing them, because the methodology involved is relatively trivial. What really made these systems successful (especially MACSYMA; this bears out the point most strongly, I think) is that they are packed full of subject matter expertise. MACSYMA is a collection of symbolic mathematics algorithms, which has been put together by really strong applied mathematicians, people who really know that field, who also happen to be fluent in LISP.

So, they were expressing themselves in LISP; and MACSYMA is the result. The person who wrote MYCIN is a doctor who is a good diagnostician himself.

What you have in these cases is people who know their field, essentially taking advantage of a computational formalism, that helps them say what they know. And it's natural enough that if they are clearheaded about it, they can get some good applications going.

Feigenbaum has made this very point - that in expert systems it is always the particular expertise that counts, not some general uniform technique.

It little becomes the AI community to say: "Look at these successes : AI technology was simply applied to this problem area or that problem area, and we got expert systems". That's not how it happened. There is no such thing as a general AI technology, which these people took advantage of. What they took advantage of was computers, and a good programming language.

Well, that may have provoked some questions when I am finished, so I will go on.

Finally, let me say something about the Fifth Generation Project. I discern, as a very friendly observer, two classes of goal in the Fifth Generation Project: one class is what you might call "software and hardware engineering". It seems to me these goals are realistic; they will be achieved certainly; they are even conservative. They are so well thought through and planned.

On the other hand, I think that the goals that you might classify as AI goals - such as speech understanding, vision and language translation - those are very ambitious, wonderful aspirations, but have a different order of difficulty, because they so much involve the unknown, with not much already in our bag of tricks to help us get there. I hesitate to say these goals are <u>too</u> ambitious; but they are of a different kind.

4.0 THE NEAR FUTURE (1991 - 2000)

In the 1990's we shall be experiencing the results of the Fifth Generation Project. We might expect that the main impact of the Fifth Generation will be what it is trying to achieve, namely, to open up all kinds of new applications of this new way of computing.

I think we <u>can</u> expect expert systems to be in general use. Once the tools are available, I do not believe that a special kind of expert - the "knowledge engineer" - will be needed to implement such systems. The point of the Fifth Generation revolution is to eliminate, as far as possible, the role of such a go-between. Today's situation, in which the professional expert is not necessarily able to express his expertise in suitable computational form, is not the model for the future. We must expect that "logic programming literacy" will become widespread.

The expert system of the near future will only superficially be super-human; it will simply be the embodiment of existing expertise as currently stored inside humans. Of course, the entrancing prospect is the possibility to scale up the speed and the size of problems, which are like what humans can cope with, but are beyond the computational capacity of the human data processing instrument.

We humans have such small buffers and such slow, if highly parallel, processors that we are strongly limited in how we can deploy such expertise as we manage to acquire in a short life time.

If we can learn how that works--learn how to express it, and how to invoke it and activate it, then we have the possibility to amplify what we already understand.

A very good example of that in today's technology is the uncanny and rather upsetting power of the best chess playing programs. The underlying process performed by all of the current chess playing programs is elementary and uninspiring, mere look-ahead in the tree of moves, and evaluation according to some quite understandable plan of weighing the features of the configurations out on the horizon, and then backing up those values in the minimax way. That's not a very deep idea, but it just happens that the scale on which it's performed is such that it's already sufficient to give a hard time to some of the very best human chess players. There are recorded examples of international grand masters finding it difficult to avoid defeat in some of the specialized endgame situations in chess, when the machine is simply playing in this open, easy-to-understand way, but on a huge scale and at enormous speed.

So, the poor human is faced with something which in principle he too could do, but which is being done on such an enormous scale, that there is a difference in degree in performance - the "order of magnitude effect".

All of that, it seems to me, might be brought about if we just extrapolate a little bit current trends in all these different fields. Especially interesting, it seems to me, is the prospect of a low-cost personal work station with all of the different capabilities that we might look for in the 1990's. It seems not unlikely that we shall each have as a personal possession something like a world library - a Library of Congress. A small shelf of optical disks, much like today's personal collections of phonograph records, would be enough to store it.

5.0 THE DISTANT FUTURE (2001 - ?)

Many people associate the year 2001 with the title of the popular film by Stanley Kubrick and Arthur Clarke in which the talking computer HAL develops a catastrophic neurosis and sabotages the mission to Jupiter. This kind of "realistic" science fiction seems not too different from the sort of rational speculation needed for looking ahead at the more distant future.

I am sure there are people in the audience who are much better placed than I to speculate. But it seems to me that we can now discern two longer term trends that will reach some sort of culmination not long after 2001.

There are already people who are investigating the fabrication possibilities opened up by genetic engineering - in which protein structures would be constructed according to programs that are in the DNA, just as they are in nature. The idea would be that we too could exploit the genetic coding and use it as a programming medium and assemble structures down in that scale. Such ultra large scale integration is the natural culmination of present trends, and it is nature's own technology. She has had much experience with it, and our brains and nervous systems are compact, complex, powerful devices built entirely in this way.

I see no reason at all why we shouldn't be looking for a direct modeling of neuro-physiological systems. The rate at which the experimental work is now proceeding in the medical research centers around the world is really quite astonishing. Last year's Nobel Prize winners, Hubel and Wiesel, have shown us some amazing things about the way the vision system works, in animals and presumably humans.

There appears to be a systematic structure in there that looks very familiar to designers of computing equipment. I think that given another two or three decades, we should be very far along in this understanding of actual natural systems, and that we will be able to reproduce them to some extent.

We should also expect the interfacing of artificial systems with our own: supplementary prosthetic devices for enhancing what we already have. Thus we might see extra memory modules, enhanced vision and hearing, and auxiliary processing units for direct access to external information resources, dictionaries, and so on. We are today seeing medical technology accomplishing many kinds of mechanical prostheses. We are beginning to be able to think of devising prostheses also to information processing functions as well.

Finally, let us try to think ahead to what intelligent computing might do for important problems which are extremely large or extremely difficult (or both) and which we now can't do much about. Detailed models of the world economy or the world ecology; spoken natural language translation in real time. Obviously, the long-range goals of the Fifth Generation Project bear upon these. And obviously, these goals will be reached. The only question is how soon. I think we can each elaborate for ourselves the speculations about what that might mean for the way life is lived, and what it might mean indeed for peace and harmony between different peoples.

I would like to conclude by saying that even though there is some sort of a language barrier between you and me, I have never had a happier and more fruitful three weeks than I am just concluding here in Japan, language barrier or not. Perhaps we don't need the automatic translator quite as badly as some people say we do.

Thank you very much. If there is time for questions, I would be happy to try to answer them.

MODERATOR: Thank you. The next: question and answer.

MR. FURUKAWA: I want to ask you about your thought to combine logic programming and functional programming. I think there are three issues; that's my question.

One is at rather philosophical level. And I think you may have some deep consideration why you need to combine logical programming and functional programming.

And the second level is the notational level. We need some kind of notational device to combine.

And the third level is the implementation: How to manage these two different ideas.

MR. ROBINSON: Thank you. Perhaps I can take them in reverse order.

The way I think we would want to implement a unified system would be to design an extension of existing function programming notation from the reduction semantics point of view, where you understand the computation process in terms of a collection of rewriting rules, which are looking for matches for their left-hand sides, whereupon their right-hand sides are replaced there; and just that is done.

There is a natural parallelism there, because many rules can find matches for their left-hand sides all at the same time. And so, if you think of the replacement being done at all places possible, you get a natural large "grain of progress". Then, if you can find a place in there for the logic programming process, you have the implementation plan, at any rate up to within details.

And I propose to make that happen in this manner, namely, to introduce a set expression in addition to the normal expressions of function programming, which are basically applications of functions to arguments. By a set expression I mean the normal mathematicians' notation with the curly brackets: the set of all x such that P(x). That is not a functional application. It has a different semantics. But you can give perfectly simple straighforward replacement rules for such expressions. And the replacement rule for such expressions essentially is to replace one of the goals in the condition part them by the right-hand sides of clauses which unify with it.

So, a given set expression is replaced by an expression saying "union of several set expressions", one each for each distinct resolvent, as we would normally think of it in a logic programming context.

If you introduce that replacement rule for the set expression, and supplementary rules for working out the details of the union construct, you find that you can harmoniously make the logic programming process happen inside the reduction of set expressions.

So much for implementation.

For notation, I think the same idea of extending the normal function applicative notation with set expressions gives a beautiful programming notation. And I am not alone in advocating it. As it happens, David Turner is now doing that with his own function programming formalism; so is John Darlington.

The function programming people have seen the need to add the set expression to their notation. It's a very natural notation; it's the sort of notation that one uses mathematically, intuitively, on paper. So, there is no problem, it seems to me, about that. It's a good move from the notation point of view.

Finally, from the philosophical point of view, I think the motive for going this way is, as always, some sort of understanding of what one is doing; how thinking works. And for that, one needs some features of one's model, which include certainly simplicity and elegance and power. This is going back all the way to Frege now, who had those criteria before him when he designed the _Begriffschrifft_. I think it's a matter of looking for forms of expression that we find natural in our own thought-processes, and representing what we do as nearly isomorphically to the way we think as possible.

These are the forms of pure thought. If you look at logic, what is it giving you? two things: abstraction, application of functions to arguments, and that's all. The set expression is really abstraction, _par excellence_.

And so, there is very little going on in that model. Application and abstraction are really the two main things that are in the notation, and I believe that makes it an extremely simple but powerful model for all of thinking. So I want to do it that way.

MR. SUWA: My name is Motoi Suwa from ETL.

Although I understood that you do hate AI, I don't really think that you hate AI. Because you give us your foreseen of Fifth Generation Computer Project, and you pointed out that the expert systems will be achieved in all areas. But you said that the "expert systems" propaganda is misleading. What do you mean by "expert systems"?

MR. ROBINSON: Well, I don't know whether you have that idiom in the Japanese language, but in English you have the notion of using the quotation marks in a rather sarcastic way. So, if you wish to mock something, you can put quotes around it.

I think another device is to preface it with the prefix "so-called".

Now, I apologize for the sarcastic quotation marks. I will erase them from my transparency. I didn't really mean to mock _expert systems_ so much as the idea that we have "AI technology" or "knowledge engineering" to thank for them. I will stand by _those_ quotation marks! I like to stimulate a debate, and it's fun to put one's position more strongly perhaps than one really feels it.

I am simply pointing at something that gives me disquiet. I don't know what fields you all come from, but all of us here probably represent a number of different fields which are very ancient: logic is a very old subject; mathematics; physics, chemistry, biology, and so on. Now there is a certain dignity to good science, which involves one in being faithful to criteria of modesty and testing ideas and being very systematic in one's expositions, and so on, which I value very highly. And I think so does everyone else who has ever been involved in that cultural tradition.

As I watch the field of AI, and its literature, and its practitioners, I find that there is not much respect for that spirit. There is an exuberant energy and an infectious excitement, but there are also a lot of careless and half-completed results being published. It's a vibrant, youthful, chaotic field.

So, while I want to identify myself with the quest of artificial intelligence, (I think it's a magnificent adventure) on the one hand, I don't, on the other hand, want to be associated with some of the style that I observe among at least some practitioners of it.

I think in fact that AI perhaps hasn't advanced as well as it might have, if more attention had been paid to some of the niceties of the older disciplines. And so, when I have an opportunity to say this in public, I do so; my intention is to try to improve the situation a little bit, to raise the standards of practice in AI.

There is a lot more AI goes on in the popular press than I would like to see. I suppose that's inevitable. The public is interested. It's an exciting subject. But if you look at, for example, the history of physics, if you look at the careers of, say, Albert Einstein, or John von Neumann, you find that they were very reticent and very careful about what they would say to the newspapers. They took great pains to underplay what they were doing, and not to hype up the excitement level in the media.

It's quite plain that such sober reticence is not the prevailing style in AI, at least at present. But I don't hate AI; I love it. I just want it to be better than it is.