RECENT DEVELOPMENTS IN MODERN COMPUTER LANGUAGES

Panelists:

Dr. Hendrik J. Boom, Concordia University
Albert K. Christians, Gibraltar Life Insurance Co.
Brian J. Fortier, Cologne Life Reinsurance Co.

PANEL

NOTE FROM THE EDITORS:

We have added a number of footnotes containing explanatory and other comments as they occurred to one of the panelists, Brian Fortier, while he was reading the transcript of the discussion.

Please note that all footnotes are Brian Fortier's.

MODERATOR (ARNOLD SHAPIRO): Al is going to start off and he says that he is going to say things that are controversial.

AL CHRISTIANS: Well, these other gentlemen will be able to correct me. Probably the audience will be able to correct me also. I'm under the impression that the biggest recent development in computers is the language in this book (Wegner). It is the Reference Manual for the ADA Programming Language, U.S. Department of Defence. You can get your copy from the U.S. Government Printing Office for $5.00 and you can get it reprinted exactly as is in hard cover for about $20.00 - it's not copyrighted, anyone can publish it who wants to.

Before this, another book (by Wegner) appeared which was a preliminary introduction to the language. It didn't include all features, as stated in it, it was not final. The book stated that the language was being finalized as it was being written and therefore you'd better wait for the next edition, but the next edition hasn't shown up yet. But you can read this book to get a very appetizing introduction to the language. It doesn't go into great detail on a

lot of things, and a lot of things it leaves out entirely, but I'll tell you first about some of the good things. Basically all of the good things are in the book by Wegner and all the bad things are in the Language Reference Manual. The first thing is that ADA is a ...

GOTTFRIED BERGER: What's the name?

AL CHRISTIANS: ADA. ADA is named after Lady Augusta Ada Byron, later Countess of Lovelace. She was the programmer of Babbage's analytical engine, actually she was the world's first computer programmer. She thought of subroutines and recursions and all these things which are really hard to figure out and to implement on a steam driven device but she threw her heart in it.

The first thing about the language is that it is based on PASCAL. If any of you have experience with PASCAL, it will rather help you if you ever get to ADA. It's a strongly typed language. That means that each location in the computer's memory, each variable in the program, has a particular type and it is absolutely impossible, according to this book (Wegner's) anyway, to refer to the contents of a variable by the wrong type. You don't have the ability, as you have in FORTRAN, to put some really bizarre integer into a variable and print it out as a character, and it spells "mother". You've got to deal with everything exactly as defined. There is only one way to treat any variable - there's no way to let array subscripts get out of bounds - there's no way to do all the programming tricks that make some programs very hard to understand, very hard to fix when you decide to change them, and so on.*

---

* At least, the better known ones - obscure programming will not be abolished by language fiat.

There's type checking, that is to say, whenever you assign a value to a variable, the types have to match - there is no way to get around that. The machine does a lot of checking for you and because the programmer can define his own types, this gives a lot of checking on the correctness of programs.* For example, in actuarial calculation, you might say you calculate an actuarial present value which is a probability times an interest factor times a dollar amount. You can make each of these things separate types. If you have an assignment statement with a present value on the left side, you'd better have something that's of the present value type on the right side. You can tell the machine that when you multiply these things together you get the type "present value". And if you leave one of them off, it's going to be some other type and the machine is going to flag it. You're not going to be able to use the wrong type. If you use the type facility, you can do a dimensional analysis, just as you can when you're doing physics problems - on your programs it's going to catch many mistakes.**

---

* This also causes some problems: types are not always well defined.

** But not much more so than a compiler that does not check type conversion. For instance:

```
"type DOLLAR is INTEGER, range -10,000 .. 10,000;
 A,B:  DOLLAR;
 A: = B + 100;"
```

is valid, even though "100" is not directly type DOLLAR.

Types were designed with the FORTRAN I and II compiler as an aid to the compiler - not the program writer - to determine the machine level presentation of an item. It is chilling to find this idea being used as a pair of manacles. The justification of strict typing seems to be that otherwrise the "stupid" programmer, "playing with sharp tools" might cut himself. In its original setting, as an introduction to programming, this is reasonable. Otherwise it has the chilling tone of "I know what's best for you".

In any case the program writer can obviate type conversion! A file record is typed only while it's being written or read: once on the medium it is only so many bits or bytes. Therefore, to reinterpret a given set of bits with "invalid" types, per PASCAL or ADA, write them under one record definition, then read them back under another one.

It also checks types between subroutines, so that programming in lots of little pieces is facilitated, because the compiler checks a the types that subroutine returns match up with the types the program calling the subroutines is looking for and similarly for procedure parameters going the other way. It comes with a programming support system that checks all those things for you at compile time, too. You don't need to run the program to find these errors you can't compile them.

I'm sure everybody here has run a FORTRAN program where the arguments and parameters didn't match and you got really odd results. With ADA this can't happen. The compiler is going to check it all for you. No problems.

One thing that ADA does, is an improvement over some of the other procedure-oriented languages like PL/1 and Fortran: say you have something like A=F(x). In PL/1, for example, A and F can be records, or they can be arrays, F can also be a function. But you have a problem: in PL/1 or Fortran a function can only return a single value, it can't normally return arrays or records.* In ADA, as in APL, functions can return arrays or records as results. You can have A=F(x)*G(x) where F is a function that returns an array and G is a function that returns a record and the * is also programmer defined. You can build more abstract programs. The programmer can deal much more conveniently with aggregates like records and arrays

---

* This is true for IBM-implementations of PL/1. The 1976 ISO/ANSI standard requires the return of arrays/records by functions.

in ADA.

The machine aspects, as I said before, are completely hidden. You can't find out, at least not according to this book, how the machine is implementing any of this - it's all hidden.* You just deal with the source program and the data that you define. ADA comes with a good programming support system that's essentially with data base manager for all your different modules; it tracks which modules affect which other modules; if you change one module, it's going to tell you every other module that you will have to recompile to keep them compatible with each other - great support for large systems built in many pieces!

Then so much for the "semi-final" design. This book, which is actually the final reference manual, came out. It tells us what the language does all of a sudden there are ways to get around virtually all the restrictions that were originally put in, so it's still possible to write bad programs in this language.** In fact, it's probably possible to write worse programs in this language than in any other if you really wanted to. You can turn off the type checking. You can drop right into whatever the machine language is. There's a statement which calls a built-in subroutine and uses, for arguments, the op-codes and operands of a machine language instruction for whatever machine you're running on.*** The subroutine executes the machine instruction.

---

* Except for some PRAGMA's.

** It's <u>always</u> possible to write bad programming!

*** This is the LINE pragma, which is machine dependent but could compile in a non-ADA language, which could be assembler.

This way you can drop into machine code wherever you want to. You can tell the machine at what addresses to put in instructions or data, how to represent the data - all the machine features are accessible to the programmer if he decides to use those parts of the language. So you can get all the machine-oriented problems back in.*

The language is moderately difficult. The Language Reference Manual says a lot of things that aren't in the Wegner book. There's a glossary of terms, which has 47 terms in it. Programmers of more conventional languages would probably have no conception of what 10 of these mean and about ten others are a little bit different but could be related to PASCAL. There's a syntax definition in the Language Reference Manual - it's got about 150 different definitions of syntax, so it's a pretty big but not a giant language. ADA will be a little bit harder to learn and to write - it's not for the amateur programmer. With APL, PL/1 subset, FORTRAN, one could start programming in a day. It will be much harder with ADA. You have got to be a little bit more serious about it to do that.

There's no automatic conversion. It's a strongly typed language, the types in assignment statements have to match; that means that we are basically back to the days of FORTRAN 2, when if I is an integer and B a floating point, you can't have I*B and so you will wind up gnashing your teeth.

HENDRIK BOOM: You can ask for conversion.

---

* A mixed curse. The primary reason for these things is the incredible "verbosity" required for doing things in a high level language that can be done in one or two machine instructions.

AL CHRISTIANS: That's right, you can ask for conversion, but you have to remember to ask and it matters how you write your constants - if you put a decimal after them or not, things like that.

Basically, the main drawback of it is that it's not really designed to be a general purpose programming language. Since it's a modern language and it was developed by a large and thoughtful committee of people, or several large and thoughtful committees of people, it does everything anybody could possibly want, but it is not supposed to be a general purpose programming language. It's basically for imbedded processes. We had a brief discussion on those this morning - controlling hardware, controlling weapons for the defense department, airplanes, rockets, missiles, sewage plants, whatever you want. But it's not for general purpose programming - the input-output facilities they give you for file access, writing reports, etc., are kind of limited.* You have to write a lot of your own code to do things which you could do more easily in other languages. The compensating advantage is that it's easy to make this code reusable by making it into a free-standing package - the language supports free-standing packages pretty well.

I guess that's all I have to say about it.

BRIAN FORTIER: May I ask you a question about it?

AL CHRISTIANS: Yes, go ahead.

BRIAN FORTIER: I haven't read that manual yet, I've read the preliminary report. Do they maintain the process handling the

---

* An understatement.

SIMULA-like or GPSS-like approaches?

AL CHRISTIANS: You mean tasks, separate tasks?

BRIAN FORTIER: Well, not separate tasks, the process handling. You can start a task and schedule other tasks to begin at a given time - I think there's enough in there that you can build a simulation model quite easily.

AL CHRISTIANS: That's right. There's a communication, they call them tasks. You can have separate tasks, there's a communication mechanism for synchronizing the tasks or not synchronizing. I don't think there's a timing for them, you can get them together at particular program statements: you can make sure this task is at this statement and that task is at that statement. Then, after they have rendezvoused, they each go their separate ways until they decide to shake hands again.

I think this will support your Jackson-structured programming a little bit better than Jackson has done it, with a little bit less of Mickey Mouse work to do it than in Cobol. Has anyone seen Jackson's book on program design? He talks of co-routines. ADA pretty well supports co-routines.

HENDRIK BOOM: I guess it's my turn now.

GOTTFRIED BERGER: Could you please tell me where ADA is implemented?

HENDRIK BOOM: Well, let me tell you that story, the history of this thing. I first ran into this project way back when I was getting my Ph.D. and dropped in at Waterloo, where I discovered my thesis supervisor walking around with a bunch of xeroxed typed pages called 'Strawman'. Apparently the U.S. Department of Defence had

decided to do something towards a uniform language for all their so-called imbedded applications and they put together this document saying what they would like in a language and handed it around for criticism and suggestions. After they got the criticism, they would be putting up a "Wooden Man" which was based on the criticism and would be more solid. Then they were to go on to the "Tin Man", "Iron Man" and the "Steel Man" - each a more precise specification as to what this language would have to do.

Now I was told how they got the original specs. They took some people who were involved in programming in the Army, Navy and Air Force, and probably the Marines too, and they locked them in a room and wouldn't let them out until they came up with specifications. And these were pretty bizarre specifications to start with.

But over the years, the specifications improved and they put more money into the project. About 1968 (that's about four or five years later), I met somebody from the Department of Defence who said that, so far, for unifying all this programming effort in a single language, the Department had spent 4 million dollars on the project. He said this was incredibly cheap: "we never spent this little money on any project yet!" In fact, he said, it was hard to get budget approval because they needed so little. Their annual software budget, at the time was something like $2 billion a year, I think, so this really was peanuts - for them; for the academics and such, who were starting to get money for criticizing these things, it was an enormous amount such as had not been seen in years. Anyway, after the specifications were "hardened" and they had gotten down to "steel" they started letting contracts for language designers to

design a language to these specifications.

Now there are a number of mistakes in the specifications because people at the time were not farsighted enough to see the implications of what they were asking for and any such defects in the specifications resulted directly in defects of the language. And there is no point in talking to the language designer and saying that's a defect - he will put back the specs and say, "that defect has to be there, I know it's a defect". This is a funny process. There was some feedback from the tentative language designs on the specs but it wasn't much. I think it was a revised "Iron Man" or "Steel Man",* but it wasn't quite as far-reaching as anybody would really like.

Of course they wanted to get the thing out.

And after that there were, I think, four design teams with tentative proposals (Green, Red, Yellow and Blue); two of them got knocked out and the other two came with further proposals. These last two were written and put together in Boston (Red) and in Paris (Green). The one in Paris was officially awarded to Honeywell in Minneapolis and Honeywell immediately contracted it out to the people in Paris that they had designing languages for them but, you know, pro forma it was an American company who got it.

After that, the language that was designed in Paris was the one that finally survived and you can see remnants of that in the fact that when two processes communicate they have a "rendezvous".

Now, what was this language originally for? Basically most of

---

* Steel Man II

the software the Department of Defence had was controlling weapons, controlling bombs, and things like that. And they wanted a better language than the ones they were using, which were things like COBOL, a DOD design to begin with, and FORTRAN and all kinds of home-grown products (such as JOVIAL and NELIAC) and mountains and mountains of assembly language, all of which were incompatible and everybody starting building a new weapon system started out to design a new programming language. So the DOD thought that this was silly, there was too much effort being expended here and that was the start of their motivation. Well, on the way, they decided they wanted to pick up all the nice things of high level languages and they sent out letters to all the well-known programming language gurus in order to get advice on things and a lot of it was good and some of it was very, very bad. Anyway, the main advantage of ADA that I see is that it does look like a sort of state-of-the-art programming language - the state of the art being measured around the beginning of the 1970's. That's when the design effort started and that's when it started solidifying. The good thing I can say about it is that it looks as if it exists and it's going to exist. There are lots and lots and lots of systems languages, some of which are better than ADA, some of which are worse; some exist on just one machine and never get passed on to anything else. The main thing about ADA is a political factor and that's that the Department of Defence, which has enormous purchasing power, is going to insist that that language be available in all the computers they buy. So, it's likely to be available, likely to be as available as FORTRAN.

BRIAN FORTIER: Or COBOL.

HENDRIK BOOM: Or COBOL.

AL CHRISTIANS: In all the versions, will it be compatible?

HENDRIK BOOM: I don't see that all the versions are going to be compatible. They tried for that, but there are inherent limitations.

AL CHRISTIANS: They still decreed it.

HENDRIK BOOM: They decreed it, yes, they decreed it. That was one of the requirements in "Steel Man", so... Incompatibilities are against the rules, of course, but how are they going to check? People implementing it don't say, "Gee, I have a slightly better way of doing that, let's hope nobody notices I improved it."

BRIAN FORTIER: Bench marks?

HENDRIK BOOM: Bench marks? Yes. But you can always add things and keep them secret - until it's passed the test.

MARK HOROWITZ: What is the relevance of ADA for actuarial work?

HENDRIK BOOM: I don't know. Maybe somebody who is closer to actuarial work will answer that question.

LISP might be more relevant but ...

AL CHRISTIANS: One thing is the wide availability of ADA; another is that the language supports well the use of libraries of functions. It's possible to build portable software, extremely portable software, extremely sharable functions. We have this in actuarial circles now with APL. Different time-sharing systems have pretty much the same APL packages and software can circulate around. You can get something from somebody else's system and it will work on your system because APL's are all pretty much the same. Because of

the mandatory standardization the same thing can happen with ADA.

The other advantage is that, as a 1970's language, it is more modern than anything else that's widely available.

BRIAN FORTIER: The real relevant thing we have is something I was mentioning just a while back. If you're doing any simulation work at all it has some of the capacities of SIMULA, you can basically build yourself an event-driven model in this language and that is probably the most significant thing for actuaries.

I might as well put my word in at this particular point, I don't think very much of ADA. It's merely a blown-up version of PASCAL with some things built into it for the benefit of the Department of Defence. And I can't conceive of making much use of PASCAL under any circumstances.

PASCAL is interesting in that it's trying to attend to one of the great difficulties of all modern programming languages - and that's data. People have been spending an awful lot of time writing clever programs to express algorithms in, and have been slighting the data. The only languages that really attend to data with any kind of attention are PL/1 and COBOL. They are the only current ones besides PASCAL which allow you structures that are meaningful in the most common cases we come across.

I should mention, incidentally, that I come from both sides of the fence, I have acted as Data Processing Manager and as Vice-President and Actuary and I have done my programming for computing cash values, reserves and premiums as well as for trying to process the master record. And, looking at the whole company's operation, ADA is insignificantly meaningful in any company's operation. As for the

actuary, possibly he could use it as a computational language if for some reason he does not like PL/1, FORTRAN, PASCAL, ALGOL or APL. PASCAL will probably be just as available as ADA and probably earlier and I am quite sure there will be more software available in it for quite some time, now that Apple has decided to put together a PASCAL compiler and, especially, since Wirth has basically given away the PASCAL compiler to anybody who wants to write a P-code executor on any given machine. There is one manufacturer who is making an ADA chip so you can now have an ADA processor in ROM on an Intel chip.

VOICE FROM THE AUDIENCE: Intel?

AL CHRISTIANS: Yes, Intel.

HENDRIK BOOM: It actually turns out to be three chips but...

AL CHRISTIANS: I thought it was three IC's on one chip.

HENDRIK BOOM: Well, that's a new one. I haven't heard that one before.

BRIAN FORTIER: I seem to have said enough. I think I see a question coming up ...

GEORGE CHERLIN: I haven't heard the language FORTH mentioned. I don't know enough about it but there was a quite nice magazine article featuring it. Apparently that's used a lot on these embedded processors, like sewing machine chips and washing machine chips, because it will work on a very small CPU. Apparently you need only a few hundred bytes to do something meaningful with it.

BRIAN FORTIER: Called FORTH?

GEORGE CHERLIN: Yes, F O R T H. The man who developed it said he got the name at the time when he was limited to five-character names and this was the fourth - F O R T H - version of it.

BRIAN FORTIER: Is this a mini-computer language?*

GEORGE CHERLIN: Yes, well it seems even smaller than mini. Apparently, you use it on little tiny chips and appliances.

HENDRIK BOOM: Yes. FORTH is basically the following. There is a collection of subroutines which can be called. A FORTH routine consists of just the addresses of the routines to be called; an interpreter calls them one after another. The FORTH routine, however, looks just like a machine-code subroutine because it starts with a real machine instruction that calls the interpreter. It also has a mechanism for reading in words, looking them up in a table, finding out what subroutine corresponds with it and calling that subroutine. That's about all the basic routine FORTH does. One of these subroutines puts FORTH into a special mode in which it stores the address of the subroutine instead of calling it when it reads a word. In this way you can use it to create new FORTH routines. There are also subroutines that put machine code into memory, so you can create new machine code routines, too. That's basically what FORTH is. You can start with a very small piece of it and build a larger system. That's the way it originated. What you tend to get is a programming language which is extremely error-prone, because anything you do wrong simply becomes another program, different from the one you intended, and usually in a bizarre way. Most of your instructions directly generate code and if you leave a "then" out but you've got

---

* The normal implementation is micro-computer, sort of a micro APL in its way. In many ways FORTH can be looked at as a package to convert a computer into a programmable desk top calculator, but with computer size.

an "if", suddenly it will do something bizarre that bears no relationship to what you want.*

It has insufficient error checking. But it runs. Its advantage is that a program is just a list of addresses. It has to be extremely small and, therefore, tends to be pretty fast too.**

BRIAN FORTIER: You can get the wrong answer faster then.

HENDRIK BOOM: You get the wrong answer faster and it's sort of more compact than the assembler code because subroutine call instructions take space - addresses don't take as much.

GOTTFRIED BERGER: Would any of you comment on the following:

I think it would be highly desireable if the tendency would be towards one reasonable language which can be used on any computer; particularly, it would be very nice if we had one higher language which would be available on the Apple computers and the home computers but also on big computers. Now, I have no idea whether PASCAL is a good language but it's not, yes?

BRIAN FORTIER: In its place. In its place.

GOTTFRIED BERGER: Yes. Now I have the impression that PASCAL is the coming language because I heard today that Apple supports it, then IBM will support it in its new home computer, then I heard Hewlett Packett will replace its internal language with PASCAL, so I

---

* But if you handle the language and machine as a programmable calculator and test each line before incorporating it in the program -- a la APL -- you can program quite efficiently. And FORTH runs faster than BASIC or APL.

** The language puts a premium on the programmer using efficiently a language close to the machine.

think PASCAL is the thing to come.

HENDRIK BOOM: I guess that sounds like my question. PASCAL is certainly growing by leaps and bounds and, I think, for a while it was promoted mainly by academics who were interested in writing compilers. They could take over versions of this thing that Wirth wrote and take his recipes and sit down and write a compiler for it by taking the existing thing and modifying it. This made it very easy to spread.

It also had very good press behind it because it was advertised essentially informally as a structured programming language and it came out at just about the time of the whole hubbub of structured programming. Now, I mean, it doesn't matter that it is a structured programming language: most of the things it has were already in PL1 and ALGOL 60. Because it came out at the time the structured programming hubbub came about, it got the benefit of the publicity, even though there had been other languages for at least twelve years that would be capable of supporting structured programming nicely. Now PASCAL spread among academics, and when something spreads among academics the students get it and once the students get it, they go elsewhere and want it. So it starts spreading by word-of-mouth.

There are troubles with PASCAL. The first trouble is that it's good for writing small programs, but it's not so good for writing medium-sized programs and it's really bad for writing large programs, because it doesn't have enough structure to organize very large programs.

I was in a place that did a lot of numerical calculations - the Mathematical Centre in Amsterdam. Among other things, they are re-

sponsible for maintaining one of the large mathematical subroutine libraries and when they got their new CDC Cyber they said, "let's go on writing ALGOL 60". Then they discovered that, on a machine that was twenty times as fast, ALGOL 60 ran at the same speed as on their old machine and they said, "My God, we can't use this compiler." So they turned around and said, "What can we use?" And they tried PASCAL and after about two or three years of trying that - it takes about that long before you know what the limitations of the language are - they pretty well had to give up using PASCAL as a main language to carry out their subroutine libraries. The reason was that it is not possible in PASCAL to write procedure or function which takes a variable bound array as a parameter so that you can pass different size arrays to it within different calls.

So, if you have a matrix inversion subroutine, you have a matrix inversion subroutine that inverts one size of matrix. Now you can parametrize that, but not by a parameter, instead by "manifest" constant, so that if you want to have a large program and you have one size of matrix to invert, you can copy the code in and have it compiled - it's sort of tailored for that one size but you can't use the same subroutine if you have two matrices of different sizes to invert in the same program. You essentially have to make two different copies of the subroutine.

Now this became intolerable - there has been some stuff done about it in the standardization, but still, this is the kind of limitations you get. Another kind of basic limitations you get every place is that in data manipulation you find that there are two classes of values in the language. There are first-class values and

there are second-class values. With the first-class values you can do anything you want, you can return them as function values. With the second-class values, which are usually of user defined data types, you can't do everything you want. There are restrictions on where you can use them. This gets in the way. After about two year's experience with the language you start to encounter the limitations. The danger is that, as a straightforward programmer, you may live with the limitations and never know they are there, have trouble writing big systems and never know why.

So, I don't like PASCAL. It's a language I use when I don't have a better alternative. Many machines don't have better alternatives.

BRIAN FORTIER: This is generally one of the good reasons for using any language, that it is all you've got. I'm quite sure more things have been written in Fortran because that's the only compiler you've got when you would like to use something else.

Frankly, I'll express one criticism with ALGOL and all of its descendents, whatever they may be (I think the only exception is PL/1): despite the fact that yes, they can support structured programming and, really, they can be written top-down, in form they look like bottom-up languages. It is very hard to read an ALGOL program of any size. What you do is to turn to the very end and try to read back until you find where the main routine begins. Then you read back a little bit further until you find where the main sub routines are, and so on ...

HENDRIK BOOM: Well, you have to know the trick to start with the end.

I find it fairly easy because I sit at a terminal when I do it. It's very easy to say, "go to the end", and I look at the last lines and say, "Oh, that's what it's doing?" Then I say, "What's that subroutine?" I say, "Go to the beginning and find me this name". I look for the subroutine and the first place that name appears is where it's declared and all the other places are calls so it becomes very easy to find with an average text editor.*

BRIAN FORTIER: But with the average text editor, what do you do if all you have is the listing?

HENDRIK BOOM: Oh well, if all you have is the listing, you have to look for it just like always.

BRIAN FORTIER: I find that actually FORTRAN is much easier to read that way, particularly when written properly, because your main routine is here and it's short and the data are not particularly big and here are your main subroutines and here are your smaller subroutines. It's a matter of linear search if it's been written properly. Any language can be poorly written.

HENDRIK BOOM: I have to admit I share a bit of your objection. I think it's an objection I can live with easily, especially because some of the other languages have very difficult objections.

GARY MOONEY: Just like your views, you guys are kind of depressing me.

HENDRIK BOOM: Well, the world's a depressing place.

GARY MOONEY: I would like to find out, first, what languages

---

* Provided the language does not support forward declarations!

you do like, and, secondly, what languages do you think will be important over the next ten years? I realize there are different languages for different purposes.

BRIAN FORTIER: What do you mean by that second part of the question?

GARY MOONEY: I mean FORTRAN, PL1, ALGOL, PASCAL and ADA will be very important over the next five years, and APL, I should mention that. Because there are many, many users of every one of these languages and, if you come across the real nuts who have the compiler, you are going to find out all kinds of reasons why they are the best languages in the world. I mean, they are going to persist and lots is going to be written in them and there are whole places of people who write nothing but COBOL; I pity the poor people, but they write that. The whole shop will just not consider anything else.

VOICE FROM FLOOR: Did you include PASCAL in that list?

BRIAN FORTIER: I included PASCAL in that list: there are an awful lot of PASCAL nuts. They seem to think that it's the greatest thing since peanut butter, really. It's a mixture of COBOL and ALGOL 60 that has come about as a teaching language and, it is true, the nuts do look at small programs. You look at any example and it's a lot of small programs, small subroutines and you look at the nice, easy algorithm and forget about the fact that 90% of any decent program is input-output which is almost impossible to do well in PASCAL.

HENDRIK BOOM: I have a sort of an answer ...

BRIAN FORTIER: Sorry, I forgot to mention my own favourite language. Given that I do scientific work, it is APL; and given that

I do data processing work, it is PL/1.

HENDRIK BOOM: I sort of agree with his list of important languages in the sense in that certain languages will be very widely used, widely available, but a couple of other important languages are LISP and SETL which is the one I mentioned in my talk this morning. SETL, by the way, is available, if anybody is interested. The implementation is available; its transformation techniques are going to take awhile.

VOICE FROM FLOOR: How about BLISS?

HENDRIK BOOM: BLISS? I don't think it will be a major important language.

VOICE FROM FLOOR: Except if you have a DEC 20.

HENDRIK BOOM: If you have a DEC 20 the same optimizer for BLISS is eventually going to be used in other languages too.

BRIAN FORTIER: BLISS is available in the whole DEC line.*

HENDRIK BOOM: Yes, but it's a slightly different BLISS on each machine.

BRIAN FORTIER: Not so that you could notice.

HENDRIK BOOM: Yes, so that you could notice.

My favourite languages, on the other hand, the ones I use regularly, or would use regularly if they were available, are ALGOL 68 and LISP. ALGOL 68 is a new ALGOL. It's essentially a new language, a different language from ALGOL 60. It's not just an up-date or a revision, it's a completely new language. The designers sort of pub-

---

* PDP20, PDP11 and VAX.

lished their first report on what this language was going to be in 1968 (hence the name), but they didn't quite finalize the design until 1975 - there were a lot of revisions and improvements and really fine-tuning of the details. I like the language and I would not really willingly switch to any other language. The trouble is that hardly anybody has implemented it. It's available on a CDC Cyber, in an expensive implementation, and there's a slow one on a 370. Because of that, I ended up using LISP, which is available, incompatibly, in most places, but I don't really care about the incompatibility. I carry my own implementation with me. That way it stays the same from place to place.

MARK HOROWITZ: How reasonable will it be for a collection of actuaries to design specifications for a language and have the Society commission a set of programmers to write a language for actuarial use?

BRIAN FORTIER: Writing a compiler if you know precisely what you want to do is a relatively simple task these days. The big problem would be deciding what you want to do.

AL CHRISTIANS: You're probably better off deciding what specific types of applications you want to do, whether they are numerical or data processing or whatever, and finding an appropriate language for that type of problem.

There is not going to be a language which is the answer to every problem. If you expect that it's possible to build a general purpose language, you're going to be disappointed. You are better off with a language designed specifically for problems like the one you have.

AL CHRISTIANS: The actuary does numerical work, data processing

work, simulation work, and often collates the results of different kinds of computer work; it's hard to figure what he needs. You have to put in a lot of different features to give good coverage of what we want to do.

BRIAN FORTIER: The real problem is that we just don't have a big enough machine to put a real general purpose language on. When you consider that the human skull contains several thousand times a CPU's maximum memory banks' worth of information how in the world is a measly computer going to cover everything that a bright actuary might want? In any case, a computer language is a tool. If you care for your tools you're not going to use your jackknife to tighten screws.

HENDRIK BOOM: A comment on ACT, the APL based programming language: there is a guiding philosophy to it. You don't have a thick manual to consult and say, "Let's see, what was the name of that subroutine again?" There are 278 names and they don't make any sense.

BRIAN FORTIER: It sounds like the COBOL manual.

HENDRIK BOOM: Yes, that's a problem with many special purpose languages. Now, my impression of the proper way to go about it is to start with a language that is very general, this time not in the sense I had this morning that it has features to do everything, but that it has a general framework of conventions as to how parts of programs will communicate. What you will do then is build your special purpose language by defining the new data types and new operators that you want to work with in a special purpose language. Now, there has been some work on these "extensible" languages. The one

that I happen to favour, I mentioned it earlier, is ALGOL 68. ADA has some properties like that, although I think ADA has too much in it to start with - it will get in your way.*

That's the direction to go. APL is a lot that way too. APL doesn't allow easy type checiking - I mean, in APL you can implicitly define how you're going to represent your data; you do that in comments and define the operators, but there's nothing which checks that you're really using them the way you wanted to use them. That's very important, not just to use operators, but to have something checking when things go wrong - that's as important to the language as that things go right when you do the right things; things should complain when you do the wrong thing.

And so you need an extensible language in which you can define what you want to do and define what you want not to do and have a check.

Now, there are a few of those around - ALGOL 68 to a fair extent, ADA a little bit. Mark Rain in Maine is putting out an extensible language, MARY, that looks good. It has no precedence rules, like APL. Well, the thing to do is to pick one of those and add to it what you want.

FRANK REYNOLDS: I'm afraid we're running into something I'm going to discuss on Saturday. The problem that we are running into in Actuarial Science is that there is a lot of demand for a nice simple language in which you can express actuarial problems. Everybody's

---

* Incidentally, without the data checking this sounds like FORTH.

got the same basic set of problems - generating premiums, cash values, dividends, reserves, a very limited number of categories of things. We also have, for many things, a nice, simple looking standard notation and, for any of us reading the notation, it's very simple to understand and communicate across language barriers and everything else. However, as soon as we start dealing with something as dense as a computer, we are immediately into the problem that the present notational system isn't perfect by any means, but does contain, and permit, a very large number of subtleties which are internationally considered obvious. But if you try and translate these into a computer, they drive you up the wall. There have been a number of papers trying to simplify, for printing purposes, the notation. All of them have an appendix at the back, indicating how we manage to get this simplified notation into the computer. It is very interesting to note that in pretty well all these papers the authors have tried to get something that is easy to type and have virtually given up trying to get something that is easy to get into the computer, because they can't translate the subtleties that are present in the standard notation. The notation is extremely complex from the point of view of trying to analyze it; yet it looks so trivial.

BRIAN FORTIER: In other words, our ancestors did a good job. I'll add something to that. At least in the work that I have done in computing premium rates, when you get beyond the classical reserve formula the notation is completely useless. Almost invariably, when I have got down to what I really want to do with premium definitions, I have gone right back to q and ω and i, and that's the only way I could express things.

FRANK REYNOLDS: There is a certain amount in what you're saying, no actuarial notation is going to be able to handle the tremendous flexibility that is sometimes necessary under a Gross Premium Valuation.

BRIAN FORTIER: Well, a Gross Premium Valuation, if done properly, cannot be expressed by the classical actuarial notation. It is, without elaborate extensions, still inadequate for Pension Plans. There is just too much to be taken into account that the symbolism just can't even begin to deal with.

The standard premium functions don't reflect the actual methods used in the office.

The standard commutation functions still reflect the computational capacities of 1900.*

AL CHRISTIANS: I think that if a notation is standardized and adequate it shows that the science using it has reached its terminal stages. If you have a growing Science you're going to need a new notation all the time. There are other fields where people publish a paper and define all their symbols in the first couple of paragraphs and there is no standardized notation. Mathematics has its standard notation but it's so large that it's almost useless; if you try to do that actuarially you might wind up with the same thing.

GEORGE CHERLIN: On the topic of new languages I think it's

---

* Notation tends to relate to computational practice. Perhaps the whole point is that computational practice has far outstripped the old notation. If so, a new notation is not worthwhile: either it becomes a programming language or it is inadequate - and there are quite enough programming languages already.

worthwhile mentioning the language "C" for completeness. It comes from the Bell System and Western Electric, and it appears to be headed towards implementation on personal computers, along with an operating "UNIX" system that might compete with CPM hence the designation C.*

BRIAN FORTIER: Similar to C, although it is not quite a language, is also RATFOR, which is basically C adapted to FORTRAN so that you can write FORTRAN programs in what looks like C in a structured way. I find that very useful, I write RATFOR quite regularly.

GEORGE CHERLIN: Apparently C was never marketed by Western Electric. It was made available for internal use and it was contracted for by outsiders who wanted to use it and they have sold a lot of licenses and made a lot of money on it.**

HENDRIK BOOM: The trouble with C is that it looks like a high level language; but all the operations, and the way it thinks internally, make it basically an assembly language. It's like an assembly language for the PDP 11, in fact, with if-then-else and while-do and things like that in it and it is often very difficult to read.***

BRIAN FORTIER: RATFOR is a little bit better than that.

HENDRIK BOOM: I know, RATFOR is based on FORTRAN which is, whatever its limitations, a coherent language.

BRIAN FORTIER: And imposing this "C" structure on it makes it

---

* C is the successor to B [CPL].

** C is spreading rapidly along with UNIX in the large micro area.

*** But all computer languages are ways of recasting assembler. Those that make for greater distance are less used.

into a very nice language.

VOICE FROM FLOOR: Is APL growing, are there proposed manuscripts that are being studied or criticized?***

BRIAN FORTIER: Depends who you talk to. I've heard it said that APL is a dead language and, also, that APL is impossible to learn.

HENDRIK BOOM: I've heard it said that APL is the best thing that ever has been invented in programming languages and people should stop inventing languages ever since. I don't believe that. I don't believe either of these statements.

BRIAN FORTIER: I agree.

AL CHRISTIANS: Last spring Computer World had an issue in which writers were expressing both of those opinions. I think that the last, and most reasoned, opinion they presented was that for 10-15% of applications APL was the best thing in the world but that you were looking for trouble if you tried to use APL on applications outside of that 10 to 15%.

VOICE FROM FLOOR: Are they going to introduce structures in APL similar to PL/1 structure?

BRIAN FORTIER: That has been done already.

VOICE FROM FLOOR: It has been done?

AL CHRISTIANS: This spring. Scientific Time-sharing has it now.

*** ISO/ANSI

BRIAN FORTIER: I.P. Sharp has it too. A structured APl-implementation is now three years old and it has basically an ALGOL structure and in APL computationals called APLGOL, without GO TO.

HENDRIK BOOM: That's a big advance.

BRIAN FORTIER: HP3000 built that thing.

HENDRIK BOOM: The other interesting thing is that I think people working at compiler optimization are almost at the point now where they can compile APL instead of interpreting it.* I think the basic techniques exist and it has to be tackled.

BRIAN FORTIER: You can use a Burroughs 6700; the B 6700 APL does compile on the spot and re-compiles if different binding forces a re-compile.

HENDRIK BOOM: The thing is that being able to compile instead of interpret means that, if you happen to have programs that don't fit well into the matrix calculus that APL imposes, you will still get a reasonable execution time.

DAVID ERBACH: I'm surprised by the implicit assumption everyone is making that any of these languages is really appropriate for much of what goes on. There are certain types of problems, - 10% or 15% - that APL or something like LISP are very good for.

But, for the majority of everyday programming you do, the issues are simpler. You have certain specific problems. For example, you want to be able to get information into the machine. That means you will want to be able to design screens efficiently, preferably so

---

* The problem is binding, not optimization.

every individual person has his own custom-made screen. You want to be able to check that, as the data goes in, it's appropriate.

There are few enough of these processes that you could usually write general software specifically to create such subroutines. Something like that would get rid of at least 90% of programmers.

BRIAN FORTIER: Right. I'll explain. I make use of an HP3000 and that's precisely what HP3000 has as part of its software. HP gives you a package that allows you to design a screen and enforced very rigid checking on input data before it's accepted into the machine and then it gives you a set of very good data base accessing techniques. So, basically, your whole program is a string of calls to these routines. Get data from the screens, put them in the data base, get data from the data base, put them back on the screen, whatever, and HP makes it very simple to create the data entry programs which is the bread and butter of this kind of operation.

AL CHRISTIANS: If you talk to IBM, they'll tell you that automatic programming is something that they are pushing for quite a bit. They say that if you have their system, called DMS, on one of their small machines, you can have one programmer, with one person who is going to be a user, write in one or two months what would normally take five programmers two years. They get productivity of thousands of statements per programmer per day. They still need a programmer to do a little bit of the work, but IBM is pushing automatic programming - everybody is working on automatic programming. I think the actuary may be one of the last to be converted to automatic programming because his work is mathematically algorithmic; algebraic programming languages are a natural way of expressing some

of the solutions.

BRIAN FORTIER: Yes. For programming you are doing for yourself.

HENDRIK BOOM: Perhaps for some kinds of algebraic programming they could take a system like Maxima, which MIT has developed for symbol manipulation. (It continues to advertise and, as far as I can tell, it doesn't make it available to outsiders - I'm not sure of that.) This is basically a symbolic algebra system in which you can do things like the integral of enormous expressions and square roots and other things so that you say "integrate this formula" and it goes away and chugs for awhile, then tells you there is no closed form or else it gives you a page full of symbology which is your integral, worked out.

BRIAN FORTIER: It would have to be rewritten for an actuary.

HENDRIK BOOM: Of course it would have to be rewritten to fit your notation.

BRIAN FORTIER: Is that LISP-based?

HENDRIK BOOM: I think the thing is LISP-based, but I don't know.

BRIAN FORTIER: I know it has been doing things like that.

FRANK MYER: These arguments over which language to use sound as if there are several carpenters, each of which has been trained to use just one tool, are arguing and each one of them is claiming that his tool is the one that is good for everything.

BRIAN FORTIER: No, I don't think we're arguing that. Now let's identify what's going on. I work in an environment where I have maybe six programmers on a machine and a lot of users but I have all

kinds of C.P.U. time available to me. Henk here works at a University and his computers are, generally speaking, assaulted by ten thousand students who have little programs that they want to compile and run. We have entirely different views of what is appropriate. We have to. I mean, he's interested in as many compilers as you can get out with the best diagnostics you can find.

HENDRIK BOOM (aside): I'm lucky where I am now because the department has just bought itself a Vax and they haven't got themselves any users for it yet!

BRIAN FORTIER: This is a normal requirement for a university and if I offer you, Mr. University Professor, a choice between a one-pass compiler which forces the programmer to do a fair amount of work to get it to work, and which also has neat facilities for solving fairly complex logical problems versus a several pass compiler which does much better input-output work, but takes quite a bit longer to run and doesn't produce as good diagnostics for a green student, which are you, as a professor, going to choose?

HENDRIK BOOM: Well, you use whichever is good for your job.

BRIAN FORTIER: In my position, I have different aims. I will take PL/1 over ALGOL any day, just on the ground that PL/1 requires me to do much less work to get the same results as an ALGOL compiler. I don't have to have nearly as rigid a structure. I can write a program in a much more natural fashion; I can just sit down and write the task out as I am sitting at my desk.

HENDRIK BOOM: That's funny because I would choose ALGOL over PL/1 for the same reason. I would choose ALGOL 68 though, not ALGOL 60, because ALGOL 68 has the singular property that it's really hard

to get the program through the compiler, it thinks of so many things to complain about. But when you finally get through the compiler, your program runs. It's a slow compiler, I mean, I've used ALGOL 68 with one-day turnaround time but I much prefer it because, if you get your program to run sooner with a faster compiler that does fewer checks, you spend that one-day turnaround time debugging - one bug at a time and not 20 at a time.

BRIAN FORTIER: Well, ALGOL 68 would drive me up the wall, as described. My PL/1 bugs are small and easily fixed.

ARNOLD SHAPIRO: Could we have a closing statement, maybe?

BILL MITCHELL: Excuse me, I would like to get one question in.

ARNOLD SHAPIRO: Go ahead.

BILL MITCHELL: Along the lines of demonstrating, substituting demonstrations for impressions, which is one of our goals, is there any hope for a quantitative approach to choosing the language? Let's say given a certain environment - certain machines, a certain programming project - defining a certain programming language?

HENDRIK BOOM: Not unless you can state quantitatively what you want the language to accomplish.

BILL MITCHELL: Well, let's say you have a project at hand, that needs to be done.

BRIAN FORTIER: The answer is yes - and no. Yes, if you have a very clear situation that will fit precisely what a language is best for, I can then tell you what that language is. But the vast majority of problems are out in the hinterland - this language is fairly good for this aspect of the problem, but that language is fairly good for that aspect.

And, theoretically, every language we've mentioned will solve every problem.*

AL CHRISTIANS: There is something they called "software metrics", which tries to come up with numerical ratings for individual programs. It's much like the underwriting process for life insurance, you come up with numerical ratings to assess the quality of a program and the chance that it will fail but all you get is an indication of your relative chance for success.

HENDRIK BOOM: You produce quantitative numbers but you don't know whether those quantitative numbers have any relevance to what you are trying to measure.

ARNOLD SHAPIRO: Could I have a closing statement maybe? Does anybody have a closing statement?

AL CHRISTIANS: I can't come up with a short closing statement. Henk?

HENDRIK BOOM: Yes, I can, at least for my part. My view of programming languages may be pictured as follows:

I've seen trees being drawn in which FORTRAN is at the bottom and whatever language the person is trying to sell is at the top and there are lots of branches with other languages.

But if you really start thinking about it, you discover these branches come together again - it's more like a mainstream with a variety of languages that aren't identical (they're good for different things) and some side streams of really bizarre languages that

---

* They are general purpose languages equivalent to a general Turing Machine.

are way off from the rest.

But what happens in the course of time is that the ideas from the bizarre languages move back to the mainstream.* People start to see how to generalize ideas and see the pattern matching isn't so bizarre after all. It turns out to be useful for writing good input processors, it's not just something for people in the natural language field and, gradually, influences go back to the centre.

So, in my view, in ten or fifteen years - it's getting slower now, maybe fifteen years instead of ten - large numbers of the various "bizarre" features you find in different languages will find themselves into the mainstream and we'll have new things to choose from.

BRIAN FORTIER: I'll make my own closing statement. The problem in advances in higher level languages is that we're suffering from terrible indigestion with what we've got already. Until we sort this out and get a better handle on data and input-output there will be no real advances in programming languages. There are really only five "cardinal" languages: COBOL, ALGOL/FORTRAN, APL, SNOBOL/COMIT and LISP/IPL. All others current in 1980 are cross breeds.

---

* For instance, APL arrays are now becoming available in PL/1.