Computer Oral History Collection, 1969-1973, 1977

Interviewee: Morton Bernstein
Interviewer: Robina Mapstone
Date: March 14, 1973
Repository: Archives Center, National Museum of American History

Note: The text of this transcript has been heavily edited by the interviewee. Items in brackets […] have been added to the text of the transcript to improve readability and/or understanding. Endnotes have been added to correct errors or to better establish context.

MAPSTONE:

The date is March 14, 1973 and this is Bobbi Mapstone talking to Mr. Morton Bernstein at System Development Corporation in Santa Monica, California. Why don't we start this with how you got into computers and number crunching.

BERNSTEIN:

The U. S. Navy had "decommissioned" the primary reason for my getting into that branch of the service during World War II— namely to repair shipboard radar—and since I was ineligible for sea duty [because I was near sighted] I ended up as a Storekeeper [Disbursing] in a Separation Center [at Bainbridge, Maryland] where they were discharging everyone [at the end of WW II]. Among other things, I learned to operate a desk calculator, a Comptometer, and got introduced to that strange part of the world that is today known as Business Data Processing. I guess I liked it as it seemed to be a part of the world that I could understand, manage and have some control over. That probably influenced me to take a major in Mathematics when I got out of the Navy and went to school, instead of something else [like Business Administration or Engineering].

I always had some kind of interest [in computation, which is why I had decided on a major in Mathematical Statistics] but [when I started to graduate school] due to a budgetary crisis, instead of my becoming a TA [teaching assistant in the Math Department]. I was told that arrangements had been made for a possible appointment as a research assistant in the Biostatistics Department of the [then newly opened Graduate] School of Public Health because of the major I had chosen. As one of my first assignments I was handed, literally in almost total ignorance, the responsibility for running a small punched card laboratory that was supposed to be used in the statistical analysis of collected data.

We started out with one of those delightful old IBM sorters with the Queen Anne legs and a bank of counters—I haven't seen one of those in years—a key punch, a verifier, and, I think, a collator. If we needed anything else, we had access to the university

administrative office's equipment, which was a reproducing punch, a 402 tabulator that had all of eight columns of alphanumeric, the rest of the 80 columns were numeric only. I started learning how to manipulate the machines when I was given a deck of cards [from which] to get the numbers that people wanted. One day, without any warning, they were hauling out my sorter and replacing it with this very funny modern looking piece of equipment called a Statistical Sorter. I think it was called the "101." The IBM salesman dropped the manual on my desk and said, "I understand this is yours," and disappeared. By this time I had read several other IBM accounting machine manuals. When I opened up the manual for the 101, it was a complete disaster. Whoever wrote the manual had no familiarity with other IBM equipment, and all of the verbiage had changed slightly. I had the machine, a plugboard, a bunch of wires and a manual that said this thing will do all kind of glorious and wondrous things. So I took the first example [in the manual], wired up the plugboard, plugged it in, punched up a deck of cards and said, "Oh, yes, it does that! Now how do I get to the real problems that I have to solve?" The solution was a strange one. It wasn't clear that the machine always worked. It did not work for all of the test cases provided in the manual. We called in the [IBM] Customer Engineer and it was through his good offices that I discovered the logical wiring diagram stored inside the machine. That was how I learned to use it. I used to trace out logical paths in the wiring diagram of the machine and say [to myself], "Ah ha! If a card goes in and if a hole comes through at this time and picks this relay thus doing thus and such and I can also get this counter to kick over here … " Eventually I and one other guy became so expert on the machine that we were the Pittsburgh area demonstration center [on that machine] for IBM. I would get calls [from the IBM salesman] saying' "Could you have the following demonstration ready tomorrow morning? I have a customer coming." Little did I know that I was being taken, by the way. I was much too young and naive. In retrospect, I guess I should kick myself, I never even got a free lunch out of the IBM salesman for all that work.

However, it started me down the path of wondering about automatic computation, and thinking rather seriously about automatic computation, and thinking rather seriously about it. Some of us, including my graduate adviser, tried to get a movement going at the university to get some kind of computational capability beyond punched card equipment. By this time, [IBM's] CPC's were around, so we went over to the Bureau of Mines which was associated or at least physically located next to Carnegie Tech (which is now Carnegie Mellon University) and looked at their CPC. We said, "Gee, what a great, glorious, wondrous thing the university could make out of that [in conjunction] with the Engineering Department, the Biostatistics Department and the Education Department." We had all kind of wonderful dreams that we were going to build that into either a home brewed or purchased computer for the university. [Then] someone brought it up before the university senate where it died.

That was [but] one of my disenchantments with the university, so [in 1952 when the opportunity arose] I became an analyst on a university project at the Pentagon [in Washington] that had nothing to do with computing much to my chagrin and surprise [despite the fact that those who went when I did were a subset of those of us who had

tried to get the university interested in obtaining some advanced computational capability]. So I spent seven months there [in the Pentagon] until the project died. But I was introduced to two things [while] there. One was a project that reported to the same colonel up the line [as did the project I was on]. They were doing one of the early [applications] of LP [Linear Programming] using Dantzig's Simplex Method. They started out to model a small piece of the Army [a platoon, as I remember] and ended up trying to model the whole thing all in one bite. That project was a disaster. [1] However, I did become vaguely familiar with the UNIVAC at that time and that stimulated enough interest [in computers] so that several of us looked around for some education. The Graduate School of the Department of Agriculture was offering a course called "An Introduction to Electronic Digital Computers." That was where I first got any true exposure to what an internally programmed electronic computer was like.

MAPSTONE:

Who taught that course?

BERNSTEIN:

That's a very funny series [of events]. The course was originally taught by Dr. Canon [of the Bureau of Standards]. I have no idea where he is now. It was a sixteen week course, and for about the first six or seven weeks, he taught us all about the SEAC, [2] binary arithmetic and the whole shtick, starting from the beginning. Here is binary and this is what the machine looks like and it was honest-to-God absolute machine language coding. I don't know that we ever tried to check out any of those programs, but we certainly wrote a half a dozen or so. Somewhere I have some notes of programs that we wrote, but none was ever run. In the midst of all of that, the SWAC[3] computer at the Institute for Numerical Analysis [at UCLA] was in deep trouble and Canon had to depart for the west coast to try to straighten out the mess. [As a temporary replacement] they brought in some young Navy Lieutenant who had been assigned to NBS to learn about computers at Canons knee. I can't remember his name, but he was a very patient, nice guy. I'll never forget the evening when he went through the subtraction algorithm and how it works in the computer. He gave a decimal example first, and I can't remember whether he borrowed from the top or the bottom, but he did his borrowing the way I had learned it and so it didn't bother me. But some upstart in the back of the room got up and yelled, "You can't do that!" The Lieutenant turned around and said, "What do you mean I can't do that?" [The person in the back of the room] replied, "You can't borrow from there [pointing to wherever the lieutenant had actually borrowed], you have to borrow from there [pointing to the other number]." The lieutenant spent the next half hour trying to prove logically why it made absolutely no difference. That was the beginning of a great many disruptions in the class. The guy found out that people would answer his ravings so he thought he was a hero.

The lieutenant disappeared after a couple of sessions and was replaced by Ida Rhodes who is the grandmother of a great many programmers in the world. Ida was a delightful

lady, very, very hard-nosed and hard-minded in a lot of things. I'm not sure she fully understood the educational process, but it made the class very, very interesting. Ida is partially deaf and wears a hearing aid and did in those days, it was 1952. When she walked into the class room the first thing she did was turn her hearing aid off or turn it all the way down. She would [then] stand in front of the room [from where she lectured]. Several of us liked to be up close when Canon was lecturing, particularly because you couldn't hear him well in the back, and I liked to see the blackboard rather well. As time went by, I had moved all the way to the last row at the back of the room in order not to be blasted out of my seat by Ida because her voice was up at about four hundred dB. I went up to her at the break after she had been lecturing for four or five sessions and said, "It may be impertinent of me, but I would like to ask you a semi-personal question. Why do you turn off your hearing aid and therefore shout so loudly?" She said that she had discovered that if she left it on and could hear herself, her voice dropped to a low level and people couldn't hear her, but if she turned it off she knew that everybody in the world would hear her if she could hear herself. She also said one of the things that turns out to be one of the giggles of all times and that is, "You never, ever take a program to a computer unless you know it is one hundred percent correct!"

MAPSTONE:

Wouldn't that be nice.

BERNSTEIN:

Yes. Well, that impressed a great many of us. I think she had a semi-profound effect on a lot of us in a couple of ways. one was that you never write a piece of code until you've done a flowchart. The other was that you [desk] check the hell out of that code before you commit it to the computer. I don't know how many people she got to in the world, that is I don't know how wide her circle of students or prot, g, s, was, but for a while, she did have a rather deep effect on some of us. I felt very shaky about going to the computer with my first pieces of code. I can remember very distinctly going to assemble a 701 program and being very upset at the prospect that it may not run. Here was this horribly large, complex, valuable thing and I might be wasting its time.

[After taking the class] I decided that this [programming] was probably the world I wanted to get into, but how would I do that? There didn't seem to be any immediate opportunities in the Washington area that I could locate, at least not in the immediate future. When the Pentagon project ended, I decided not to go back to Pittsburgh. I looked around for a job [which in retrospect] led to a very funny incident, the true meaning of which I didn't understand until about three years later.

Before we left Pittsburgh to go to Washington, my wife wanted to see if she could locate a job [in Washington] before arrival there. I had one [a job, that is] and she wanted to [continue to] work as a [Registered] Nurse, her profession. She went to the United States Employment Service [which I'm sure no longer exists] and was interviewed by a guy who

had nothing to do with nursing. When he found out that I was a mathematician, he insisted that I come down for an interview. I told him that I already had a job. He said, "That's all right, you're liable to want a job in the future and we like to have all you guys located. If you are going to be in Washington, give me a call, we'll process it [my application] and you're in." When the Pentagon project was dying, I called him up. He said, "Great, come out to Arlington Hall for an interview." Everybody [in Washington] knew that was the National Security Agency. I drove in there and was escorted to the personnel office by a Marine with a submachine gun under his arm. We talked for a while and, finally, I had an interview with a couple of technical people. The conversation was so nebulous and vague that I was beginning to get upset. Finally I said, "Just exactly what kind of a job am I being interviewed for?" One of the guys weasel worded around and finally I said, "Now come on, lets keep to the straight an narrow and just tell me what it is I'm going to be doing."

He said, "I can't tell you."

I said, "Fine. Can you tell me what kind of equipment I'm going to use, whatever it is I'm going to be doing?"

He said, "No, I can't tell you that either. After you get in and get your clearance, then you can find out about it."

So I decided, "Gee, this is pretty vague. It may be interesting. It may be dull." Then we got into a problem with the personnel people who couldn't arrive at the right GSA level for me given my experience, so the whole thing got aborted. Instead, I took a job beating a desk calculator to death doing engineering computation because that was the closest thing that I could get at the time to what I wanted to do.

After I was at RAND for about a year, Tom Steel joined the corporation. I found out that he had been at NSA at the time [of my interview there]. What was going on [that they wouldn't tell me about] was that they were installing an IBM 701 and were looking for programmer trainees. I missed my chance to get into the field by two years. But no one would tell me, it was a highly classified thing.

Back to Ida Rhodes for a minute. She had some terribly strict prejudices. Apparently she had been a consultant to Eckert and Mauchly in designing the UNIVAC, and that was the "only real computer" in the world. [She declared] that dumb thing they called The Defense Calculator (which happened to be the IBM 701) was never going to work. Every once in a while she would let fly and rail against the 701 being a total waste of the government's money. We really didn't know what she was talking about.

MAPSTONE:

You hadn't met the 701?

BERNSTEIN:

No. Nobody had met a 701 unless you were either a) with IBM or b) with the National Security Agency which is where the first, or an early one [4] apparently went. The Defense Calculator was all we knew about it and I don't remember whether she used its number or not. But she sure was against it, whatever it was.

MAPSTONE:

Did she ever say why?

BERNSTEIN:

I don't know whether it was just a strong anti-IBM prejudice or the architecture. You know that the UNIVAC and the IBM 701 were totally different in architecture [a term that wasn't used in those days]. One [the UNIVAC] was a decimal machine and the other one [the 701] was a binary machine.

By the way, when she [Ida Rhodes] took over the course, we switched completely from talking about the SEAC and things binary to the UNIVAC. The final exam was a delight. I don't think I still have it, but in essence, it must have taken some 30 hours to complete. What she literally wanted done was (a) have several programs written for the UNIVAC and (b) have several program for the UNIVAC decoded to find out what they did. I know that we weren't all that bright. I don't know if anybody passed it. I don't remember seeing a grade for the course; I had gotten what I wanted out of it and I really didn't care much about her dumb final examination. I do remember that two or three of us who took the course were rather up tight about that exam. It was pretty horrible. The problem was that we didn't have manuals, so we had no reference material to work from. We only had our notes and some of us were in disagreement about how the machine really worked at that point in time because it was new.

MAPSTONE:

It's surprising that you ever got a program on the machine at all.

BERNSTEIN:

I never got to the UNIVAC with a program.

MAPSTONE:

It was always theoretical?

BERNSTEIN:

It was all hypothetical, at least. After I [had] worked for Atlantic Research for a couple of years, [5] I started a propaganda program inside the company. We were doing a tremendous amount of thermodynamic calculations for both propellant design and rocket nozzle design. It was just terribly nasty iterative solutions to differential equations with an awful lot of wasted motion, time and effort and a hell of a lot of mistakes made by me and three gals beating desk calculators to death with not very adequate methods for checking [the accuracy of the results]. Heaven only knows how many rotten things got all the way through the machine shop into the model stage, out to the test stand and then blew because we had made a numerical error. I have no idea. We were also doing an awful lot of [solid propellant] grain design of the geometric properties, and I kept saying, "God, there's got to be a better way to do this." I started a campaign to get some automatic computational capability installed in the company, but they wouldn't buy—they were too small.

I can remember one time conning my boss into visiting ERA [Electronic Research Associates who] I guess had one of the world's first [computer] service bureaus. They had an 1101 installed in Arlington, Virginia. I had made an appointment by phone and we went there [to see what they were selling]. It appeared that they were selling [computer] time and I figured, "What a delightful way to learn to program for real. Take some our problems and program them to run on this delightful computer." They gave us the standard kind of demonstration—Flexowriter in, Flexowriter out, lots of blinking lights—a running computer. But my boss had to go and ask the critical question, "How much does it cost?" That was the end, another aborted effort. I can't remember what they wanted, but I guess that the rates were rather ungodly. He [my boss] could look at the shop and say, "I can hire three girls for six months for that kind of money." What he didn't understand was that he was going to get four times as much work [out of the computer] as he was out of the six girls, but I couldn't convince him because he wasn't sure that we were going to be able to get the programs written and working. In retrospect, he was probably right; he had great prescience about things like that. And I got slowly but surely very, very tired of that whole desk calculator bit.

MAPSTONE:

They never went into things like CPC's?

BERNSTEIN:

I couldn't convince them to get anything like that, they just weren't about to.

One Sunday morning I was sitting in my living room reading the Washington Post and although I've never been an inveterate want ad reader, every once in a while I page through the entire paper. There was an ad that said, "Programmer trainees wanted by the RAND Corporation in Santa Monica, California." My wife was in the kitchen preparing lunch and I said, "How would you like to go to California?"

She said, "That would take me 3000 miles away from our in-laws, wouldn't in?"

I said, "You're right. I'd better go and have an interview."

MAPSTONE:

That was a good reason.

BERNSTEIN:

That was one of the primary motivations to pick myself up and get on the phone and make an appointment. So I called and made an appointment with a Mr. Armer at the DuPont Plaza Hotel that afternoon. I got dressed up to the teeth in a blue suit, white shirt, sincere tie, shined shoes—the whole bit. I drove to the DuPont Plaza, parked the car—my wife was going to wait for me in the lobby—called from the house phone and was told to come on up. I walked in and there was Paul Armer sitting on the bed like a Buddha. I don't remember whether he needed a shave or not, but he wore a sport shirt and was sitting cross legged on the bed. He introduced himself and we had about a 45 minute chat, at which point he said, "If you're interested, fill out the application and send it to me. We'll consider it." I was interested and I did. I had heard of the RAND Corporation and they had a great reputation inside the Pentagon.

Three weeks later, as I was about to leave work, the gal who was the [Atlantic Research] receptionist, telephone operator, gal Friday called to me, "Stop, I've got a call from California for you. Why don't you take it over there." pointing to a phone on a table in the lobby. Since three weeks had passed, the interview was completely out of my mind. I picked up the phone—it was Paul Armer making me a job offer. I didn't know what to do: whether to talk to him loudly enough so that the people streaming out the front door could overhear, or be cryptic. Finally, we agreed that his offer sounded perfectly reasonable and that he would send the necessary papers in the mail.

A couple of weeks after that I was off and running. When I arrived at the RAND Corporation, I was given my first task—my introduction to the [IBM] 701. In retrospect, it was a rather interesting and fascinating one. (I don't think we bring people into the business the right way anymore, but then we can't because we bring them in hordes instead of a few at a time.) I was handed an IBM 701 manual and a binary punched card. I was told that the layout of the information on the punched card was either in the manual or in an attachment that came with it, or that one or the other contained the information from which I could figure out the layout. I was then supposed to figure out what the program on the card did. I struggled and sweated and struggled and after a great deal of time and effort (there was nothing better for me to do because without a clearance I couldn't enter the building [without an escort]), I discovered that [the punched card] was a one card bootstrap loader. [6] It was a very interesting and fascinating introduction [to programming]. It assumed that you had a certain basic understanding about things, and

supposedly I did because I had been introduced to computers and some of the theory before that, but it was a heck of a problem to figure out.

I got so far [into the solution] and there was this dumb thing beginning to overwrite things that it had just loaded. I'm sitting there scratching my head trying to figure out whether it is a mistake, whether I don't understand, or that it really should be doing that. At this point I had to decide [whether] I should go on and see what happens or declare that this is a disaster and ask somebody—there was a whole lot of pride involved.

MAPSTONE:

That's right.

BERNSTEIN:

So I learned about the 701 the hard way. Fish or cut bait.

MAPSTONE:

Once you learn something that way—

BERNSTEIN:

You never forget it. But as a result of that one is now tempted to do all kinds of cute, wondrous things like trying to improve on the one card bootstrap loader. I would venture to say that quite a few hundred man hours were used up at the RAND Corporation by people writing newer, trickier bootstraps and the like.

MAPSTONE:

Everyone has been telling me that only very serious and important work was done at the RAND Corporation!

BERNSTEIN:

I think there were some exceptions to that. I don't remember the exact wording, but a memo appeared that said we weren't allowed to use RAND Corporation and government equipment on our own behalf for solving puzzles and games in order to win prizes.

MAPSTONE:

Win prizes being the key.

BERNSTEIN:

Two or three guys using either the CPC or the 701, cracked one of those big puzzles and they won quite a few bucks in the process. That got somebody upset because they were afraid that the government would hear about such things.

MAPSTONE:

Not for profit.

BERNSTEIN:

Well, not so much not-for-profit, but the straight misuse of government property [for personal gain]. Since everything that RAND had was owned by the government, there were lots of people who didn't take to that very kindly.

There were lots of fun and games [going on] in retrospect. When I got there, and for a long time afterward, I'm sure that there were no overt [programming research] program going on [at RAND]. IBM had guys like John Backus who had proven their mettle and were off doing interesting things [like FORTRAN] with the blessings of the corporation. But there were very few people at RAND who didn't have a standard applications programming assignment. We didn't have systems programmers in those days, we had people assigned to what was called the Utility Team. They were supposed to maintain and upgrade the quality of the utility programs needed by people doing the applications. We [RAND] had a closed shop for an awful long time. Nobody touched the computers or a coding pad who wasn't a member of the Numerical Analysis Department, or as it later became known, the Computer Science Department. Today, it's something else. Only the name has been changed to protect the guilty.

Whatever research got done was the result of people having a desire and a drive and being willing to fiddle either in their spare time or on their own time[7] to try various and sundry things. Much of it was known [by management] but people looked aside and none of it had any overt blessing for quite sometime that I can remember. However, a lot of fun and funny things got done. Of course, there were some people who had a higher blessing than others, but a lot of programming support development, like assembly programs, got worked on and some of the language development that was done at RAND was done sort of on the QT, because somebody had a strong motivation to do it on his own and didn't let it interfere too severely with what else they were supposed to be doing. After a while, I attained some reasonable blessing and in time research was recognized as a perfectly legitimate activity as long as the corporation could find the money to support the guys who were doing it. I got into it essentially full time that way.

MAPSTONE:

When you got into RAND, which area did you actually get into?

BERNSTEIN:

I was in the Numerical Analysis Department as a programmer trainee. I have a hard time recalling the first job I did. I remember going to the 701 in fear and trepidation with the first deck to be assembled and Jimmy Wong, who has been at RAND and SDC since 1952, literally holding my hand, taking me to the console, introducing me to the operator and saying, "Okay, now this is what you do. You drop your deck in here, making sure you have a copy of the assembler on the front and the proper end card on the back. You also have to remember to put the one card bootstrap clear memory on the front, and then you're going to get a binary deck out. We will take the binary deck and see if there are any errors in the assembly listing, then load the program and go." I don't remember whether we made it all the way through. We made it through the assembler, but whether or not we were able to run the binary deck or not, I have no recollection whatsoever. The whole thing was almost a traumatic experience because we all knew that the damned machine cost something like $600 per hour for just the bare machine costs and that didn't take into account anything else. [8] To stand there for a minute [doing nothing] was a frighteningly expensive thing to do, particularly relative to salaries in those days.

I remember the delightful mechanism we had for taking a console scoop [dump]. There was a Polaroid Land camera in a wooden box mounted on a post with a long air driven cable release. If you wanted to know what the console was doing, instead of writing it down you turned to the operator and said, "I want a console dump." He would go "click," then [climb a short ladder and] pull the film out of the camera, wait until it developed and hand you your photo of the console dump. I don't think we were unique, other people were doing this, but it was the only place that I know of where it was a regular part of the operation. The camera was positioned on the post directly facing the console to get a good image and the lights were arranged so you didn't get any glare. You could always read the console and the registers that were there.

Many years later when the 701 was long gone I was riffling through some things and came across a couple of those console dumps stuffed into an album. I pinned one of them up on a [bulletin] board for a joke with a note saying, "To Whom Does This Belong?" I stood there watching some of the younger programmers who had never seen the 701 looking at it and wondering what in God's name it could be.

I think the first real program I ever tried to write was for the integration of a set of partial differential equations representing the reentry into the atmosphere of an ablative missile nose cone. We looked at the problem and it appeared that it was going to run for a tremendous number of hours because of the mesh size required, so we decided to do it in fixed point rather than with the floating point interpretive packages that were around. We got the integration routine coded up and I got some test data from my customer in the RAND Engineering Division. I can't remember how I made sure that the program would cycle properly, but we got to the point where we were going to run a real case. I loaded the program and stood there. For the first run I was going to take output [every cycle] to see how things were progressing and to see whether it was going to oscillate or go into a loop. I got no printout. It ground and ground and after about four minutes I was becoming rather perturbed because everything had worked up to this time with the test data. Finally,

after about five or six minutes—I had signed up for a very long shot so there was nobody in the queue bugging me—I was beginning to be bugged because it was taking so long to get the first iteration [to finish], the thing was going to take about 100,000 times longer than I had wanted. I stopped the machine and took a dump and went back to my desk to ponder it. I found out that we had integrated the hell out of nothing. It hadn't moved off the first point. It was still at the boundary. Nothing had moved. I sat there greatly and deeply puzzled. I finally went to Mario Juncosa [the numerical analyst on the department staff] and said, "I've got this funny problem I don't understand." We went through a few exercises and he said, "You lost your precision. You need one more bit. You don't have it. You need a thirty-seven bit word instead of a thirty-six bit word and then it might go." We tried to devise all kind of schemes that would solve the problem other than using some kind of interpreter, because there was just no way to run any reasonable number of test cases using an interpreter that ran fifty to a hundred times slower than fixed point arithmetic. Finally, the poor guy who had specified the job got very upset and decided to abort the whole thing.

[The following occurs earlier during the interview in the middle of the above anecdote. It is moved back to here so that the stream of the above anecdote wouldn't be interrupted.] RAND was a collection point for everything. They didn't run a single programming system [on the 701 in the days before operating systems], they ran multiple [assemblers]. The floating point interpreter you wanted to use dictated the assembler you used, if you were going to use any at all. RAND had [and used] a Douglas assembler, a Los Alamos assembler, the IBM assembler, and one that Cliff Shaw wrote. That makes at least four [assemblers] that I know of and there may have been as many as five or six. Each required its own special format. The only one that came close to being a symbolic assembler was Cliff Shaw's. The others were either relative decimal or relative octal. The Douglas assembler had some of the flavor of a symbolic assembler in that you weren't constrained to absolute sequence numbers—you could insert and delete without major penalties. There were great arguments [within RAND] about whose assembler was best. Everyone had their own pet. That made for some very interesting problems when two pieces of a program were coded up by different people and somebody forgot to tell them were working on a single problem. If forgot to tell them were working on a single problem. If one [person] used the Douglas assembler and one used the Los Alamos assembler there was no way to put them together [without recoding one part]. One such incident actually occurred, but it was worse than that [see below].

[After the ablative nose cone disaster] I went to work for Bill Orchard-Hays on his Linear Programming project. My main involvement was in writing a few service routines that became part of the LP package for the 701. I also had some more interesting chores, like actually putting together the data sets for execution on the LP system and helping to write the operator's procedures for running LP on the 701. The operator's manual for LP was at least an inch thick, and I'm not sure how logically it was constructed. I don't have a copy of it. I do remember being called at 2 o'clock in the morning by one of the brighter operators saying, "I've gotten into a condition here that's not covered. What do I do now?" By this time the program had already run for three and a half hours. The LP

program was replete with check-point-restart points and what the operator wanted me to do in the middle of the night was to tell him how to back up to where he could successfully get it restarted again. I usually had no idea. That happened many, many times.

MAPSTONE:

Well, you had to be creative.

BERNSTEIN:

What you really had to be was dedicated. I'd climb out of bed, into my clothes, into the car and drove to RAND to try and help the operator get things going again or to decide whether or not to run something else.

As a result of working on LP, I got into an exercise with George Dantzig that turned out to be one of the more enjoyable things I've ever done. George got interested in this problem because someone had "misused" his Simplex method for solving Linear Programming problems to optimize a diet [by minimizing its cost]. The diet came out to be primarily peanut butter and lard and that upset George mightily.

After a great deal of effort, we created a matrix from the contents of the Department of Agriculture's handbook Composition of Foods that contained all of the information on calories, proteins, carbohydrates, fats, minerals and vitamins. We were going to solve the diet problem by maximizing the bulk while holding total calories to 2000 per day. Maximizing bulk was done so that you would have a full tummy and be comfortable. In addition, it was to satisfy all of minimum daily requirements for vitamins and minerals that were in the handbook at that time. I can't remember the exact sequence of events, but perhaps George [Dantzig] has notes on it. It was so much fun at the time we were doing it that I never thought of keeping notes, unfortunately. I do remember that one of the first diets that it produced contained seven gallons of vinegar, guavas, some swordfish and a bit of cheese, among other things. [It was limited to seven items because that's all the equations we had for the matrix.] It looked quite unpalatable—particularly the vinegar. So we threw the vinegar out of the data. It was included because it had no calories and satisfied the need for vitamins and minerals. [After we threw the vinegar out, the next diet was almost the same with ketchup in place of the vinegar.] We threw out a few other things based on that first diet and ran it again. This time we got three gallons of ginger ale, plus the guavas and the swordfish. We were quite perturbed until we realized that the ginger ale, by unit weight has very low caloric content. It was going to bring in as much as it could to satisfy the bulk. So we dehydrated everything in the matrix and ran a few more cases, which produced one diet with olives and a few other things but which was not very palatable. There was no way to force such things as milk and bread and butter and meat into the solution and they never appeared in any of the diets. We finally threw out a lot of things that one would never in a million years eat under normal circumstances. This finally produced a diet that George decided to try.

It came about like this. George's doctor apparently told him to lose some weight, which is how he became interested in this approach to the problem. His doctor agreed to supervise him if he came up with a reasonable diet. After all of our manipulations we came up with a diet that George thought was palatable. It broke me up completely. It was made up of about a pound and a half of bouillon cubes, some olives and minor amounts of other things. [Still no bread or meat or potatoes.] George was dead set on trying it. I kept saying, "George, you're going to have to drink seven hundred gallons of water to get down a pound of bouillon cubes." Each bouillon cube weighs something like an eighth of an ounce [that's 128 bouillon cubes to the pound for which you would need about 1000 ounces of water which is about eight gallons]. I kept saying, "George, you don't want to do that."

And he would say, "Well, you know, it satisfies all the needed requirements. And with all that water I certainly won't have an empty stomach."

I found the whole exercise so humorous from the beginning that I had begun posting the results of each run from very nearly the beginning on a bulletin board near my office. As word spread, people would come by to see the latest result and go away laughing and joking about it. It was one of the more enjoyable things I did in the early days at RAND. It actually went on for several months. [There were several days between each run because of the time it took to modify the voluminous input data and scheduling the machine time for each lengthy run.]

MAPSTONE:

Did it really?

BERNSTEIN:

Oh yes. George was quite serious.

MAPSTONE:

Did anything useful get done with it? Did it get into the Food and Drug Administration?

BERNSTEIN:

George may have written it up, but I don't remember seeing a paper published. [9] I don't know if he kept notes on the interim things we did in order to get to a point that he believed was a realistic solution to the problem. George Dantzig left [RAND] not long after [we finished the "diet problem"]. I don't know how much [of the reason for his leaving] had to do with our silly diet and how much had to do with his running battle with Dick Bellman and related problems, or whether he just took a leave of absence.

I gained a "funny" reputation at about this time. Almost everyone that I had done any kind of programming work for left [RAND] within a year or less after I finished working for them. [It started with the engineer who was the customer on the ablative nose cone problem.]

It wasn't long after that Phil Wolfe came to RAND. He introduced upper bounding and some other sophistications to Linear Programming, like a preference matrix and did a real [edible] solution to the diet problem. He was able to come up with very realistic diets. He gave people lists [of foods] and asked, "What do you like best?" such as milk, bread, potatoes and things like that. He could get very realistic diets if you wanted to hold calories constant and meet the minimum daily requirements for vitamins, minerals, etc.

MAPSTONE:

So it really did start up the whole thing. It became serious.

BERNSTEIN:

Well, I don't know what the connection was between the work that George was doing and what Phil did—there were other things going on in Linear programming at the time. We were trying to find discrete solutions (integer solutions) for problems that couldn't tolerate fractional solutions. At the time, [Ralph] Gomory [of IBM] was working very hard on that. Slowly, but surely, I began wandering away from LP and I became interested in programming languages like FORTRAN. There were a lot of things going on at that time. SHARE was being formed because the IBM 704 was beginning delivery. I wasn't intimately involved at the beginning. I was on the periphery. It would be several years before I became a full participant [in SHARE].

MAPSTONE:

We were talking about Dantzig and his contributions.

BERNSTEIN:

I don't know the history of how Dantzig got to the RAND Corporation. Apparently he had started to develop the simplex method for solving the linear programming problem (I'm not sure it was called that in the beginning) while associated with the Air Force, I believe.

There is no question, historically, that Dantzig laid the theoretical foundations of the Simplex Method for solving LP problems, but Bill Orchard-Hays deserves a great deal of credit for being able to implement the Simplex Method on the kind of computers we had at the time. He had a thorough understanding of what Dantzig was doing and the imagination and creativity to know how to do it. I certainly never attained that level of understanding. I understood the basic things that were going on, shuffling vectors in and

out [of the basis] and testing to see whether or not you made an improvement and the selection criteria to bring in the next vector. It was nice clear algorithmic stuff and I understood how it was done. But I never really had the gut feeling for working one's way over an [n-dimensional] convex surface. I just couldn't internalize that convex hull.

Bill Orchard-Hays took that very primitive tool, the IBM 701, even though its mean-time-to-failure was probably somewhere between twenty minutes and an hour, and developed a program that allowed the solution of LP problems requiring eight to ten hours of crunching. He thoroughly understood how to deal with that situation and created a check-point/restart capability to accommodate the very short MTBF. (The industry has forgotten about that capability for the most part. But we are getting back into needing that capability again because the size and running time of programs is exceeding the MTBF of present day machines.) The complexity of the job that had to be done and the tools that one had to build it with—a very crude assembler (most of the code was written in relative octal) and nothing as sophisticated as a link loader—posed some very nasty problems for data organization and data structures.

I don't know who invented the collapsed vector, but in order to get as much stuff crammed into memory as possible (the good old 701 had all of 2048 36-bit words [of main memory]) for the relatively sparse vectors in LP problems, Orchard-Hays implemented a version of the collapsed vector where the header told you which elements were there. The values themselves were stored beneath the header, and as a result you could store a hell of a lot more information in memory or on tape that way than you could ordinarily.

Then there were those delightful 726 tape units that had the ability to read backwards but also had the ability to destroy you before you got very far. One had to cope with the problems of reading things in and checking to make sure you had valid data so you didn't go off chasing your tail. As a result, checksums became prevalent everywhere. The [computer's] drum was used for the current crop of vectors and then one was able to read the tape forward and backwards without having to rewind it. Thus, one could go through the whole matrix in the forward direction on one pass and then in the backwards direction on the next pass, making optimal use of the machine for everything that was going on. I believe that was a monumental feat. After someone does such a thing the first time, other people can pick up the trick and transfer it to other problems, and soon a lot of other people can use the idea. I truly believe the Orchard-Hays deserves that credit. These were not dull and trivial problems. We tackled some awfully big matrices in those days. We were trying to get the program to handle matrices of 512 vectors, which for that time was huge. Once we got that far, the next logical step was for someone to ask for the ability to handle 1024 vectors. Once you reach a barrier, someone will invariably want to know why it can't be pushed and in those days a lot of people were pushing very hard.

[Increased size was only one aspect of pushing], people wanted more sophisticated solutions, such as upper bounds, preference matrices and integer solutions. I can remember working very hard with Harry Markowitz and I can't remember the other

person's name, trying to devise a method for getting integer solutions. That meant that you ran the problem to its optimal non-integer solution. Then you explore all of the integer solutions in the space around the optimal solution and find the optimal one, depending on whether you were maximizing or minimizing the functional. It was an awful way to go because it meant somebody, namely Harry Markowitz and his compatriot, had to define a cutting plane that went through the integer solution you were looking for.

This was all done by guesswork; scratch your head, run the problem again, make another guess, and so forth. In order to guarantee that you had the optimal integer solution, you had to have at least three cutting planes. Thank God someone has invented a direct way to obtain discrete integer solutions. I don't know what people would do today if they hadn't. Once all of these capabilities became known to the world, once the oil companies and the econometricians got their hand on LP, it became a standard item in the collection of things that computers were very good for. And the whole RAND effort slowly disappeared. Phil Wolfe stayed around for a while continuing development. Then he went off to IBM. And RAND, [where it had all started] was no longer the center of Linear Programming. With Dantzig and Phil Wolfe gone, when Orchard-Hays left, there was no around to take on the role of charger.

MAPSTONE:

But the pioneering work had been done.

BERNSTEIN:

The pioneering work had been done, that's right. And they [RAND] did export several versions. [which reminded me of the move from the 701 to the 704.] I'll never forget the day that Bill Orchard-Hays found out that the tape drives on the 704 didn't read backwards. His whole algorithm, the beautiful structure that he had built [to run efficiently on the 701] was to be transferred directly from the 701 to the [newly announced] 704, which was a bigger, faster, better, more reliable machine. [A large contingent from RAND] went to IBM announcement at the AIA building in the Fairfax district. I don't remember whether it came out as part of the description of the machine or as the result of a direct question, but 727 tape drives would not be able to read backwards as the 726 drives had, and Bill was completely crushed. It meant a complete redesign because all of the timing had been predicated on the ability to read the tapes backwards. It was after LP had been re-implemented on the 704, and the 704 had become a very ubiquitous machine, that the LP code got disseminated to the world and things began to die off at RAND.

We also built an LP code for the JOHNNIAC, which I had no hand in. Since the JOHNNIAC had no tape drives, the program was severely limited in the size of problem it could handle by the limited amount of drum storage available.

MAPSTONE:

Who did that work?

BERNSTEIN:

Primarily Leola Cutler, who is still at RAND. It so happened that from the time they got LP running on the JOHNNIAC [about 1956] until 1961 or there about, there was one particular LP matrix that would run for about two and a half hours and then come to a screeching halt. Nobody could find the bug in anything; the data was clean, the program was clean, the machine looked clean and nobody could find the problem. Every once and occasionally someone would rerun that program to find out what went wrong. We had a very niggling feeling that it was the computer, but there was no way to pin it down. The only way was to get to that condition was to run that problem for two and a half hours, anticipate that it was about to occur and dump everything. Everything looked fine, so we'd let it run for another two minutes and it would crash. Looking at the results one could see that a vector had gotten hashed and thrown the program into a tizzy.

In 1961, or thereabouts, one of my functions was "mothering" the JOHNNIAC. My function was as the interface between the users, who were very few by then, and the engineers and technicians. One of the engineers or engineering technicians [I believe it was Dick Stahl] was trying to increase the capacity of the drum by doubling the density [with which data was written on it]. He wanted me to write a very special test program for the drum because the existing drum diagnostics had some restrictions that wouldn't allow him to test the full capacity once he doubled the density. He was afraid of one problem: Because of the doubled density, there may be a problem in synchronizing the first and last words on the track with the clock track, because the new clock track wouldn't be as clean as the old clock track had been. So I wrote this little program that would not only allow him write junk to [the drum], read it back and check its validity, but would also allow him to do a little bit of sensitivity testing on the gap between the end and the beginning of the track. The design of the drum was such that you could start anyplace and read n consecutive words. It didn't matter whether you started in the middle and wrote them to the end or around the end. It just wrapped around. It turned out that we found a very funny bug—if you started within less than eight words from the end of a track and tried to read over the gap, you got garbage for the first three words. I said, "Gee, Dick, that's bad news. You can't double the density."

He said, "Its very strange, nothing I did should affect that. Let's put the drum back in the original condition and see what happens when we run the test." Sure enough, [after he restored the drum to its original density] if you started within eight words from the end of the track and read over the gap, the first three words were garbage. Guess where the vector that got hashed and hung the LP program after two and a half hours started? I took us nearly five years to find out what went wrong. Old bugs never die.

MAPSTONE:

And that never got tested.

BERNSTEIN:

Nobody ever thought of doing that. There were bugs like that floating around all over the place for years and years. We found a delightful bug in the IBM 704 not long after it was installed at RAND. One combination of floating point numbers when multiplied with a particular one in the MQ [Multiplier-Quotient register] gave a wrong answer. If you reversed the order and put the other one in the MQ, you got the right answer. There may have been others, but the one pair was found after an exhaustive search by Nancy Brooks standing at the console of the machine trying to figure out why she got such screwy numbers [from the program she wrote]. She had been peddling through a program by hand looking for what had gone wrong. Finally she came across this result that she had hand computed in octal and said, "Wait a minute, the machine is wrong here!" Everyone told her to go back to her desk and recomputed her result, that she didn't know how to multiply in binary or octal, that she had converted something wrong. It turned out she was right. There are all kinds of disaster stories like that—hardware was full of them.

Back to the main stream of things. There was a project going on that preceded the formation of SHARE but which was really its philosophical basis. A cooperative effort was formed in Southern California comprising RAND, the Naval Ordinance Test Station at Inyokern, Douglas Aircraft, Lockheed, North American, possibly Los Alamos and several other companies. This group was called PACT [Project for Automatic Coding Techniques]. They were building a truly symbolic compiler for a vertical language, very much like [single address] assembly language, but a bit more sophisticated and had, among other things, indexing capability.[10] From my vantage point, that cooperative effort had a great impact on the people who thought about forming SHARE [most of whom were the managers of the people participating in PACT]. They already had a cooperative effort ongoing that proved that companies [highly competitive aerospace companies at that, with the exception of RAND and the Navy] could get together and share the load and come up with a meaningful and useful product. Unfortunately, PACT I was a little late and PACT II was not terribly well received because it was on the tag end of the dying 701. Although there was an abortive attempt to transfer it to the 704, it never made it—FORTRAN was coming!

That was when the raging argument started between the vertical and horizontal languages proponents. FORTRAN was the first horizontal language, and everybody [on the west coast] said, "Oh, pooh, pooh, that is terrible. The notation is terrible. No mathematician would accept that; it turns out rotten code; it takes forever; and you forgot to provide some very basic and vital things we all know and love, like subroutines." Probably its major fault was that it had no honest-to-God obvious subroutine capability. It had the Computed GOTO and the Assigned GOTO, but it didn't have any equivalent way to branch in the way the computer did. We had all learned how to code a branch and link. The standard convention on machines without index registers was to load the location of where you are into the accumulator and branch to the subroutine. The subroutine

incremented the value in the accumulator appropriately to pick up either arguments for the subroutine or their locations and, last, but not least, the location to return to from the subroutine. When the first [single address] computers arrived, it took people a while to figure that out. Go up to a programmer today and tell him that you just took away all of the Branch and Link instructions. All that's left are the straight old branch instructions. Now I want you to create a subroutine linkage convention out of the remaining instructions. He will sweat and strain mightily before he comes up with, "Load your own address into a register and branch" because it is not terribly obvious to do that.

MAPSTONE:

And you don't have to do that today,

BERNSTEIN:

That's right, so you don't even think about the problem. When someone puts the problem to you it sounds like a very weird problem, on the face of it. And then, of course, there was the delightful TSX instruction on the 704.

MAPSTONE:

TFX?

BERNSTEIN:

TSX—Transfer and set index. Everyone adopted the convention that index register four [11] would be the one you would use as the branch and link was one of the first SHARE standards.] But FORTRAN contained nothing analogous; the Computed GOTO and the Assigned GOTO were hokey attempts. That, plus the fact that it ran compilations very slowly and that they [RAND] probably chose the wrong person to assign the first use of FORTRAN to were part of the reason that FORTRAN I got minimal acceptance at RAND. His first problem as a programmer was a rather large one. He wrote somewhere near 900 FORTRAN statements for the program. Every time he found a bug, he had to recompile all 900 statements, which took in excess of a half an hour of 704 time. He was eating up machine time like it was going out of style. He spent most of his time trying to get the entire program to compile successfully. The FORTRAN compiler still had some bugs in it. On top of that, he wasn't a very good programmer, so it's no wonder he never got the program to run, even though he successfully compiled it several times.

After that experience we decided that there had to be a better way of doing things. It appeared that optimization, at least the kind that was done if FORTRAN wasn't all that necessary. It was believed that the hard part of building a compiler was in getting the compilation of algebraic statements done efficiently. Then a few of us tried building compilers and discovered that compiling algebraic statements was only about ten percent

of the job. It was all the rest that killed you; providing I/O, data conversion, trig and exponential functions, and input-output conversion routines is were the real effort goes.

However, I did write a couple of small compilers for the JOHNNIAC and that's probably how I ended up taking over as the computer's "mother" around 1957. We had a reasonable user population then, but out of the main stream of RAND's computing, so it wasn't worth investing any more money in [the machine]. It was still valuable enough to maintain, but not valuable enough to increase the investment.

MAPSTONE:

By now, it had a core memory on it, didn't it?

BERNSTEIN:

It had the first commercial core memory. When I arrived at RAND in 1954, the machine was either about to be powered down to remove the old Selectron tube memory or had just been powered down. The new core memory was due in December of 1954, I think, and when I got there (or was given my clearance so that I had free access to the building) they were doing the mechanical modifications in preparation for the installation of the Telemeter Magnetics memory. It was a little late. Telemeter Magnetics lost a lot of money on that one. There was a penalty clause in the contract that allowed the price to be reduced up to 50% for late delivery. They had bid something like $2 per bit and ended up being paid around $1 per bit. They were late; late enough to have incurred the maximum penalty. Once the core memory was installed, it ran well. With core memory, the JOHNNIAC was at least five times as reliable as the IBM 701. But as time passed, the machine got worse. There were more and more days that when the machine was powered up, it just wouldn't run. It was like a …

MAPSTONE:

Tired?

BERNSTEIN:

No, just arbitrary. Power was always left on the tube filaments. In the morning, to bring the machine up, you turned on the B plus voltage and ran the diagnostic programs. Some mornings it just took off and ran beautifully and on other mornings, nothing worked right. Tuesdays and Thursdays were hammer test days. We discovered that solder ages [in the cold environment of the computer] and as it ages, you get crystallized solder joints (called cold solder joints). They caused intermittent failures and the only way to find them was to vibrate the chassis. On Tuesday and Thursday mornings, one of the technicians with rubber hammer in hand would run the diagnostic program called "The Leapfrog All Orders Test." The purpose of the test was to execute every one of the machines instructions from every memory location, given enough time, of course. With

the test running, the technician would open each bay and tap each chassis with the rubber hammer several times.

To understand how the technician knew when an error occurred as the result of his hammering, you have to know that the JOHNNIAC, unlike the IBM machines, had a built in facility for making sound. An amplifier and speaker was attached to a machine toggle in the instruction decode matrix. That toggle in turn could be connected to a single instruction installed specifically for making sound called HOOT, any single class of instructions, such as branches or multiplies, etc., or to every instruction, controlled by a switch on the operator's console. The switch was usually set to intercept all instructions. Every program made a characteristic sound and anyone familiar with the program could tell from the sound it was making what the program was doing.

The Leapfrog All Orders Test would begin at the low end of memory execute its instructions and then move itself a bit higher in memory and repeat the sequence. Since this program was run every day without fail, the technicians became very familiar with how it sounded when it was testing instructions and when it was moving itself to another location in memory. If the sound deviated from what was expected or stopped altogether, the technician knew that the chassis he had just been hammering on had a problem—usually a cold solder joint, but sometimes a bad tube. In either case, the machine was stopped, the chassis removed and replaced with one that had been recently repaired and tested and the All Orders test continued until every chassis had been thoroughly hammered and no other failures occurred. And that was how we debugged the machine on Tuesday and Thursday mornings.

MAPSTONE:

That is lovely.

BERNSTEIN:

Yes. The problem was that once you'd located the failing chassis, which of about 40 tubes and/or 500 solder joints had failed? If no replacement chassis was available, it could take hours to get the machine back on the air.

MAPSTONE:

I believe the machine was beginning to phase out in the late 1950's.

BERNSTEIN:

Yes, but it wasn't decommissioned until 1966. But one more major modification was made before they killed it. They added [eight] typewriters and buffers so that JOSS could be built. That started in '61 or '62. There are a whole lot of interesting stories that I'm not sure I ought to put on tape. There are personalities involved and as long as the people are

still alive, their feelings might get hurt. I've told other people the stories, but not for publication or recording.

MAPSTONE:

Can we talk about JOSS?

BERNSTEIN:

Let's go back to early language development things.

MAPSTONE:

There's one thing I'd like you to explain to me before you do that. I'm not sure I quite understand the terms vertical and horizontal.

BERNSTEIN:

These terms were coined in those early days and have disappeared into history and lost meaning in terms of programming. A vertical language is when instructions lie one after another in vertical sequence on the page—like Clear and Add, Store, Subtract, Add, Multiply, Divide.

PACT was analogous to assembly language in that you could only perform one instruction per line. You could add two things together with a two address language. You could multiply two things together. You could only perform one instruction per line. You could add two things together with a two address language. You could multiply two things together. You could store something. You could load something. You could read or write. But you could only do one instruction per line. That makes a language very vertical because you read it that way. FORTRAN, on the other hand, says Y = (X**2 - Y**2)**2. I read that horizontally. That's the basic difference. Since every language today is horizontal, except assembly language, there's no particular distinction.

MAPSTONE:

OK. Good.

BERNSTEIN:

But language development didn't really get going until FORTRAN II came along.

Everyone had his pet assembler on the 701 which caused great incompatibility problems. Herman Kahn, the great, wondrous Herman Kahn, came to three different people in the [Computer Science] Department separately, I being one of them, and said, "I have this little problem (written on the back of an envelope, as he was wont to do) that should be

programmed for a computer. This is the way it ought to be done and this is what the data looks like going in and this is the kind of answers I expect coming out."

"Fine, Herman, I'll see if I can get it done for you." Because at the time I was deeply involved in doing things on the JOHNNIAC and it was more accessible, I coded it up for the JOHNNIAC using the floating point interpretive package. He also went to two other people and gave them each a problem. One did it in one of the 701's floating decimal interpretive packages and the other did it in one of the floating binary interpretive packages. And then Herman said, "And now, the output of that one is the input to this one, etc., and I would like them all put together."

And we said, "What? When do you want it?"

"I want it tomorrow. You have them all written, they all work, and now all you have to do is pin them together. Right?"

"Wrong! We have to rewrite two of the three and somebody has to make a decision."

MAPSTONE:

Well, what a mess.

BERNSTEIN:

Well, it was a minor disaster. I think Herman was mildly upset when he discovered what had been done, but nobody said that these were three pieces of the same problem; that the output of one was the input of another and that all of the parts were going to be put together at a later time. There was the possibility of manually converting the data between the parts, but that was less than desirable.

Which reminds me of another thing I got involved in around 1957. It was a war game done in the basement at RAND. It was so highly classified that it was done in a vault behind a heavy steel door with a combination lock. The responsibility for opening the vault in the morning and locking it at night was rotated among the senior members of the team. All of the work was carried on inside the vault were we worked for weeks on end. It was decided that no new programming would be done for this exercise because of the cost and time delay and because there already existed programs needed for the job. However, this could only be done using a manual interface between these programs. One of the programs ran on the 701, but RAND no longer had a 701, we had a 704. However, Douglas Aircraft had three 701's. The program that ran on the 701 was an air battle model, the one that ran on the 704 was a model for the fallout from nuclear weapons, and the overpressure damage assessment model ran on the JOHNNIAC. All of these program had been written at other times for other purposes, but they were all applicable to this particular exercise.

At three o'clock every afternoon all of the strategists for the blue team—a retired admiral, an Air Force colonel and a couple of RAND civilians—would get together and lay out an air strike. All of the parameters for the air strike would be entered onto keypunch forms and be keypunched so that by five-thirty or six o'clock the person who was to run the air battle model would pick them up and take them home. At one o'clock that night he would go to Douglas and run the air battle model on one of the Douglas 701's. The first time he went to Douglas to run the program, he forgot to get a pass to allow him to remove the output of the runs. He could take the cards he had brought in but not the printout of the results. That got fixed quickly.

At eight o'clock the next morning he showed up in the vault with the ten variations of the air battle runs. Someone would roll a ten sided die and decide which of the ten runs would be used, which told us which planes got through to the targets and which hadn't. From the planes that did get through, we took weapons on board, the aim point, the altitude, and all of the other parameters that determined where the bombs fell and fed them into the 704 fallout model program. It produced fallout and overpressure information that was then hand coded and keypunched and run on the JOHNNIAC to get the damage assessment of those targets that were hit. By this time it was around two-thirty, three o'clock and we had the results from the previous days strike and could now plan the next one. That exercise ran for days and days and days with a large number of people involved.

If for any reason, Douglas couldn't sell us time that night or the 704 was preempted by a higher priority job or the JOHNNIAC was down, nothing got done that day. There was no way that any of the programs results could be hand computed. That was one of the ways that computers were used in the early days in war gaming situations. The whole exercise was unbelievable.

I can't remember specifically how I got into programming languages. Having mothered FORTRAN and realizing that it was an interesting effort, but, in my opinion, not a very viable tool was one entree. I had decided that there were some things we could do better. At the time there was also a lot of clamoring to open the shop at RAND. There was resentment against the programmers being the only ones who had access to the computers while the people with the "real" problems didn't. It was decided that we might be able to supply a service [that would let the engineers and others at the computer] if we could invent a clean enough language that they could use and not destroy a lot of machine time. Perhaps give them access to the JOHNNIAC, which wasn't very heavily used by that time.

I started by writing a little compiler that turned out to be an interesting exercise and a valuable lesson for me. I decided on my own to write a compiler that was aimed at the engineering aides at RAND who did desk calculator computations. I took their desk calculator work forms and tried to let them lay out a job as they would for a desk calculator and with minimal transformation get the computer to do most of the work. I wrote a short manual and concluded it had some terrible shortcomings. It wasn't very

successful, and I decided that there wasn't a very big market for it. People weren't really interested in doing things like that. So I decided to build a compiler that other people could play with and wrote one that approximated a normal compiler, attempting to solve some of the nastier problems, like subroutines. I don't know if I was successful or not.

There was a tremendous number of ideas floating around at the time. Around that rime, Al Perlis had come through RAND and talked about how to really do all this. He stimulated a lot of people to at least think about it if not do something about it. I got deeper and deeper into the whole mystique of language design and compiler construction, and began to follow the first international effort—IAL—rather closely. RAND, SDC and some other organizations decided that the International Algebraic Language, which was eventually called ALGOL 58, was something of interest. Even though it had some flaws and ambiguities in it, at least it looked like a reasonable improvement over FORTRAN. From it came JOVIAL at SDC.

At about that time I got involved in SHARE, the SHARE FORTRAN Committee in particular, which raised some very interesting problems that I didn't understand until I was exposed to the world of IBM. People, including myself, would ask for modifications and additions to FORTRAN. I can remember Bill Heising [of IBM] standing up on many on occasion and saying, "That is contrary to the philosophy of FORTRAN." And that was the end of that. We wanted a simple dictionary [from the compiler] to use for getting more readable memory dumps used for debugging, so that you knew what type of variable things were. That was contrary to the philosophy of FORTRAN. We wanted a READ statement where all you gave it was a list of variables and that didn't need a FORMAT statement, in two forms: One with just a list of variables that were given the values of the numbers as they came off the input stream; the other needed that dictionary so that you just gave a simple READ statement with no variables and the input cards would contain the variable name, an equals sign or some kind of punctuation followed by that variable's value. Both were contrary to the philosophy of FORTRAN. In fact, IBM claimed that this form of READ was impossible to do. The fact that I had already done it on the JOHNNIAC and a lady from Los Alamos had already modified their FORTRAN to do it and was offering it for free to the world was immaterial. IBM was saying, "No, no, no, it can't be done." So it was turned off.

MAPSTONE:

I presume that Backus wasn't one of the IBM people who were saying FORTRAN could only do this, this, and this and couldn't do other things.

BERNSTEIN:

Backus was way out of the loop, off doing other things by then. Though he was one of the original six or eight creators of FORTRAN, he was no longer involved.

MAPSTONE:

Oh, I see. It was now a corporate venture.

BERNSTEIN:

Yes. We are in the 704 era now and Bill Heising was the manager of languages or some such and the other IBMer involved was Bob Bemer. In retrospect, we [the SHARE FORTRAN Committee] were all bushy-tailed, bright-eyed, capable guys who could go home and do anything we really wanted to, all by ourselves or with very few people helping. By today's standards, some very simple work was done, but it would get done. Nobody worried about how the resources got allocated because we weren't in a era where every minute of the computers usage had to be accounted for and charged to someone's contract or billing number. If you wanted to sneak a little machine time at six o'clock in the evening or at two o'clock in the morning, or sometime on the weekend, nobody really cared. The whole business of who uses what resources to get what done just wasn't an issue and an awful lot of "unblessed" research got done because nobody said. "Hey, you used two minutes of machine time, who are we going to charge it to?" We weren't bugged about things like that.

Obviously, IBM was in the business world and we weren't. We didn't have a proper appreciation—and in retrospect, I still don't—for how IBM decided to allocate its resources. FORTRAN could have been markedly improved in those days because there were a lot of bright people with a lot of bright people with a lot of bright ideas. Maybe they were contradictory ideas, but some of them could have been incorporated, and although the cost to IBM may have been high, it would have enhanced their leadership role. But IBM never acts—they react. That was the stance they took; a very negative one and I could never quite figure out why.

ALGOL 58 came along and excited a lot of people, but neither ALGOL 58 nor ALGOL 60 was really accepted in this country, because by that time FORTRAN was really rolling down the road. SHARE had promoted, or helped promote FORTRAN II rather heavily and then along came FORTRAN III which in retrospect was a rather idiotic attempt. There was a problem in that there were certain things you couldn't do in FORTRAN. For instance, it didn't permit you to incorporated machine language subroutines easily. The loader was kind of crude and the allocation of storage was absolute instead of relative. The FORTRAN Monitor System didn't have a link loader in it and the monolithic FORTRAN program was loaded into core at the same place every time. The problem was how to get machine language subroutines loaded without a link loader. That eventually got solved, but at this time, it wasn't. So FORTRAN III was born and it allowed you to do much of what JOVIAL did, dropping into assembly language by adding a statement that allowed you to pop in and out of assembly from FORTRAN, which everybody knew was a disaster. But it did create a wondrous argument in SHARE. I can never remember which side of the argument Frank Wagner was on. He got up in the SHARE General Session and made a great impassioned speech as he was wont to do when he takes his stand.

MAPSTONE:

What were the arguments for and against?

BERNSTEIN:

The argument for it was that it made the language a hundred times more flexible: I can now do direct I/O, real time or near real time processing, or code inner loops that had to be highly efficient. Remember, optimization was only global in that it optimized the assignment of index registers. It did not do any real global optimization by removing redundant things from the program, such as removing a "fixed" statement from inside a loop. It wasn't being done at that time, even though the knowledge of how to do it may have existed and local optimization within statements was new. The result was that people would look at the code produced by FORTRAN and say, "My God, I can take out twenty to forty percent of the redundant assembly language statements that the damned compiler inserted, such as the extra instructions it would throw into subroutine linkages. If I could write that part myself, I could certainly improve the efficiency of the executable code." There were people who could make a good case for doing that. They would write a FORTRAN program, check it out and run it forever, so they wanted efficiency in the object code. There were people against it because it made the code non-transferable. In fact, it may make the program unique to someone's particular 704, which would be a disaster, at least a philosophical disaster. So the argument raged. I believe that FORTRAN III did indeed get blessed eventually but it died almost as soon as it was born, and besides, the 709 was coming.

MAPSTONE:

We were discussing some of the pros and cons of FORTRAN III, and

BERNSTEIN:

As I say, it died a natural death. I'm sure there are still people around who remember it, but I can't remember anybody ever getting any great use out of it. That was the era when we made the transition in SHARE from "We will do it ourselves and distribute it" (IBM's, primary responsibility in their relationship to SHARE was to provide coffee for the meetings, run the SHARE distribution agency, distributed all the programs, worry about all the papers and catalogs, getting good copies keeping archival material, and things like that) through the transition that said SHARE will design it and IBM will implement it (that was the SHARE operating system and the fiascoes that came with that) to the era where SHARE designed nothing and IBM designed, implemented and distributed and sold whatever it was they thought best for the marketplace. That transition goes through the period 1957–61 from when SHARE did it all, distributed the SHARE assembler, SAP for the 704 and all of the utility programs, to the design of the SHARE operating system that was obviously too big for SHARE cooperatively program.

That is where SHARE philosophically, I think, began to come apart at the seams. The level of commitment was not at the same level of dedication that it had been in the past when you could get people assigned to do something. It took a lot of time for that [do it yourself approach] to die. We still did get some cooperative efforts going but they didn't have any true impact on the world. FORTRAN—an IBM product—was a bundled, not sold, piece of software. I don't think anybody in SHARE gave much thought to rewriting pieces of FORTRAN and distributing them on their own. There was a great deal of hassle in SHARE at the time about the true meaning of distributing programs in FORTRAN because when the 704 came and SHARE was formed, the first honest-to-God standards of the computing world were set, the standards were: "You code for distribution in the SHARE assembly program—SAP. You will comment in the following ways and the write-up will have the following format and descriptive material and there will be something called COMMON." COMMON [12] was born out of the SHARE program distribution standard. It was the scratch [or temporary] storage that subroutines use. [Every program] assigned a COMMON block. In every write-up of a [library] program the statement had to be made of how much COMMON the program used, e.g., that from COMMON through COMMON + N was used by this [library] program. Thus we had some idea of how much scratch space each program needed [by determining the maximum COMMON block used and, thus, the maximum block the entire program needed—it was a space saver]. That's how COMMON idea was born. When FORTRAN II came into being, and sub-routines finally arrived for FORTRAN, the whole concept of COMMON was absorbed and somehow changed slightly, perhaps for the better, it's not entirely clear.

I have the strong recollection that FORTRAN III actually saw the light of day, but it died. [13] It just didn't have it—there were people who were much closer to FORTRAN because I had begun to move away from FORTRAN per se and pursue other language oriented things. I was looking at the SDC's JOVIAL. I was trying to build some small compilers for open shop languages at RAND at the time and then I got involved in the SHARE ALGOL effort in '58-'59.

MAPSTONE:

Let's talk about ALGOL.

BERNSTEIN:

Okay.

MAPSTONE:

For instance, who were the people who initiated it?

BERNSTEIN:

Oh boy. The people to talk to about that are Al Perlis, who is at Yale, Julian Green [of IBM], and another IBMer who was involved in programming languages, Bob Bemer.

MAPSTONE:

Oh, yes. He was with GE.

BERNSTEIN:

Yes. In Phoenix. Now there is a long-time veteran.

MAPSTONE:

I'd like to get a hold of him, actually.

BERNSTEIN:

There is another guy floating around here who probably started in the [programming] world about the same time Bemer did. I think he may have started at RAND or Lockheed. I see him around from time to time, his name is Bob Bosak.

MAPSTONE:

And he was at RAND or …

BERNSTEIN:

I know he was at Lockheed and at SDC, and I believe he may have started at RAND before that. I'm pretty sure Bemer started at RAND and left RAND before I got there, and ended up at Lockheed and IBM and maybe a stop in between at GE.

MAPSTONE:

Okay, on ALGOL: is that Julian Green?

BERNSTEIN:

Yes, Julian Green of IBM. He is now at Equitable Life. But I don't know if he came into the picture with ALGOL 58 or whether he came into the picture beginning with ALGOL 60. I'm trying to recall who the ACM representatives who went to the first ALGOL or IAL meeting. [14]

MAPSTONE:

Is ALGOL a European … ?

BERNSTEIN:

Well, there was a letter written by, I believe it was Rudishauser, to possibly Al Perlis through the ACM[15], and said the world was going off in a thousand different directions with this language stuff. Maybe we ought to get together, the Europeans and the Americans, and design one internationally based language that we would all agree upon, and maybe we will solve some of the world's nastier problems, particularly in program communications, if we have a common source language. We may end up getting real power in the world of commonality, usability, and things like that.

I have to back up. I just remembered something. There was another effort going on in SHARE called UNCOL: Universal Computer Oriented Language, which I got my fingers into. As a result of that, the ACM appointed a group, and I don't know whether Perlis was the chairman of the group, but I believe he was. I know John Backus was a member of the group. And I am trying to remember who the others were. Joe Wegstein may well have been a member. I know he was on ALGOL 60. I don't know whether he was on '58. Anyway, the group met in Zurich and sat down and designed a language. What they came back with was ALGOL '58. And they had done it very, very swiftly. Everyone came with their own ideas; there was an awful lot of argument and compromise. But it wasn't a terribly bad language. It had some ambiguities and some holes, e.g., they completely left the I/O out, so that was up to the implementer to decide how he had to do it on his local computer. It may not have been a full quantum improvement over FORTRAN, but it was an improvement. It incorporated an awful lot of things which you couldn't express or only expressed with great difficulty in FORTRAN. In IAL or ALGOL it was reasonably straight forward.

MAPSTONE:

Predominantly scientific.

BERNSTEIN:

Yes, definitely. Nobody of that group was a commercial data processor at all, at least not to my knowledge. It was mostly just university [types] and IBM, who was the only manufacturer represented, by John Backus because of his outstanding reputation in these things.

MAPSTONE:

Right.

BERNSTEIN:

I don't know whether CODASIL was formed before or after the arrival of ALGOL 58. I have very, very vague recollections about COBOL, because it was something that we all

laughed about as being a travesty and a joke. Nobody could be serious about something as awful as that. I can't pin down the time in my head, but there was a far better business-oriented programming language, which was the first thing that Computer Sciences had gotten a contract to design and implement for Honeywell. It was called FACT. When COBOL came along, it destroyed it.

MAPSTONE:

Oh, really?

BERNSTEIN:

The DoD [at the time, the world's largest acquirer of computers] came out and said, "Any DoD organization who orders a computer has to show cause why they don't need a COBOL compiler." And that just about put every other business-oriented language out of business. It also destroyed that wondrous and glorious thing invented by IBM, called COMMERCIAL TRANSLATOR.

MAPSTONE:

So ALGOL had a short life, too, is that right?

BERNSTEIN:

There were a lot of us, including me, who sat around and looked at ALGOL and realized that it had a lot of faults. There are notational things that could be cleaned up. There are things that they forgot to put in which could be easily added. There are ambiguities that really had to be straightened out. It didn't have any I/O. And it is not all that easy to get common implementation because there are these vagaries of interpretation about what certain things meant. And I can't remember whether ALGOL 58 was a block structured language or not, but there were some misunderstandings about the scope of variables and all the other things that you run into when you start putting together rather complex language structures.

After some early and some aborted attempts to implement ALGOL 58, which is when I got really involved with ALGOL through SHARE. We decided that ALGOL was a good thing and that SHARE was going to implement ALGOL 58, or at least convince IBM to help us. We would help specify how it was to be done, and IBM would do the implementation. Bob Bemer, Julian Green and several other people at IBM were building something called XTRAN, which was a tool that would help them implement almost any language, but in particular ALGOL. Things go to the point where it was clearly understood that ALGOL had to be cleaned up. And so the [original] group [that produced IAL] was to reconvene in late '59 or early '60. Their report came out in early '60, which is why it was called ALGOL 60, although they actually met in '59. Perlis was the chairman of the group that went to Paris to clean up IAL. That is when the name ALGOL was

invented. [16] The earlier effort was designated ALGOL 58 because the new one was designated '60. I know Joe Wegstein was there. I know Julian Green was there. I know John Backus was there because that is when Backus Normal Form (before it became Backus Naur Form) was invented. I don't know whether it is apocryphal or what but the story goes that John finally figured out [how to represent the syntax of a programming language in a formal notation] on the plane over to Paris. I don't quite believe that, knowing John. He works very long and hard and maybe he had an of inspiration [on the plane] but the [representation problem] had been brewing as a result of '58. And the difficulties they had trying to get a syntax description pushed him into working that problem very hard. There were thirteen people [at the Paris meeting] in all. Seven from Europe and six from the United States, but I can't remember any of the rest of the six. The seventh U.S. representative was killed in an automobile accident just before he was supposed to go. I can't remember who that was [William Turanski].

MAPSTONE:

It is documented?

BERNSTEIN:

Yes. They [the U.S. committee] came back and Julian Green showed up at the SHARE meeting in March following the ALGOL 60 meeting, and he quietly presented me [as Chairman of the SHARE ALGOL Committee] with a draft copy of the report written in Backus NORMAL FORM. I looked at the thing and had a hemorrhage, because we had all been working very hard on the supposition that they were going to go away and fix ALGOL 58 and make it useful and wonderful and clean and beautiful and unambiguous. What they had done was gone away and thrown ALGOL 58 out the window and started clean again. They came up with a whole new completely incompatible language. People who had implemented ALGOL 58 in any shape or form really had thrown their money away because that was no longer going to be the international language, ALGOL 60 was. The compatibility was just not there. I got very, very upset. I wrote Dr. Perlis, who was then at Carnegie Tech, a rather harsh and nasty letter about dereliction of duty and a few odd things like that. I'm sure there is a copy of it somewhere in RAND. I couldn't get it [mailed] out of the building first time I tried. There was a lady who controlled all outgoing mail from the RAND Corporation. Her primary job was to see that no classified information was sent out without proper protection and was going to the recipient who knew how to handle it. She was also the guardian of the corporate morals, ethics and language and my letter was rather harsh. I had gone to Paul Armer and said, "I am going to send this very nasty letter. Be warned. Apparently, he saw it and said, "If that is what you want to say, okay, that is what you want to say." The next morning when I came in the letter hadn't gone out. We had a minor battle inside RAND convincing them I had every right to say what I was saying, both as a member of the RAND Corporation staff and representing my chairmanship of the SHARE ALGOL Committee. Finally, it got out the door. And I will never forget Perlis' answer scribbled on the bottom of my original

letter that said in essence, "Go away, you bother me or you don't bother me." But I think that was the death knell for ALGOL 60 and for the concept of ALGOL in this country.

MAPSTONE:

Maybe the death knell of any kind of universal language.

BERNSTEIN:

The basic problem was you can't get thirteen guys together and in six days invent the world. And it wasn't the right flavor of people. The mix was all wrong. I have yet to be able to plow through the wondrous and glorious [latest] ALGOL report. But it is probably a major improvement. It is far more consistent, has far fewer ambiguities, and many well be one of the most powerful and delightful programming languages in the world. If you could only understand the description. It is a terribly, terribly complex thing and you must spend many hours studying it in order to get any appreciation out of it. But it was done by a fairly small group over a long period of time who really honed and worked the problem.

Julian Green, Joe Wegstein and I had a rather uproarious session which I didn't find very uproarious. I got rather upset when I heard the stories of how ALGOL 60 was created. It was like putting a piece of legislation through Congress rather than a bunch of scientists getting together and saying, "What's best? What's cleanest? What's purest? What really is scientifically or technologically the best possible thing, the apotheosis of our skill and our knowledge." Instead, what they did was barter as in, "If you will let me put this in, I will let you put that in. No, no. I am not going to vote for your thing if you are going to argue with me about that thing." Joe Wegstein was sitting there telling me that when things got out of order he would slam his hand down on the table and bring everybody to silence by shaking them up. It doesn't sound to me the way you design a language. You go away and the think about it for a while. Think through the problems: What is the objective of the whole thing? Who are we trying to get to use it? How would you like them to be able to use it? Are there any implementation aspects that you might take into account? Can this thing be implemented at all on the existing computer? Must it wait for the next generation? Things like that. That wasn't the kind of considerations that they made. Now they may have had all those considerations in the back of their mind. And maybe I am being over-critical, but I believe they blew it.

They just destroyed whatever first, good step they had taken by not going after ALGOL 58, as impotent as it may have been in their two year further down the pike view of the world. Who cared if it didn't have "own" variables. Nobody really understood what they were all about anyway. They probably represent one tenth of one percent of the utility of a true language. The decision that all procedures had to be recursive was an interesting concept, but not something you lay on the world before they are thoroughly and completely educated in the value and use and the need, and where you may or may not want to get into things like that. And they just came out with all this beautiful,

gorgeous—not terribly pragmatic—inconsistent language. And then they did other wondrous and glorious things. They said, we will have the publication language and we leave to the implementer the hardware representation. And that is where we had great fun in SHARE. I must have spent sixty or seventy hours sitting in on arguments, chairing a committee, trying to satisfy all the diverse points of view about a SHARE hardware representation for ALGOL 60. Up to this time SHARE had a resolution which blessed the whole thing. And it was subsequent to that that I became completely disenchanted with the whole thing. The deeper I got into it, the angrier I got, and finally introduced a resolution that SHARE withdraw its original resolution and castigate the idiots who had done that, rather soundly. It didn't get that strongly worded, but we withdrew our original commendation at ACM and the [other] parties involved were going off on various crusade to create an ALGOL international language.

MAPSTONE:

What was the SHARE resolution?

BERNSTEIN:

I think it ended up just withdrawing SHARE's original resolution. It said: We no longer bless it, we don't care. SHARE no longer has any interest in ALGOL—particularly ALGOL 60.

MAPSTONE:

But the language did go on, and is still extant today?

BERNSTEIN:

ALGOL 60: yes, people still talk about it. It may be used in a few universities.

MAPSTONE:

Is there not a language called ALGOL which is in use today?

BERNSTEIN:

Yes. In fact, there is still some ALGOL 60 compilers around. How much of ALGOL 60 they really cover is not clear to me. SHARE actually did eventually, after I gave up, produce [a compiler]. This was a SHARE effort. We decided that we didn't have much of a chance to convince IBM [to implement ALGOL]. IBM was completely antithetical to ALGOL by this time. They had gotten turned off and put most of their interest in FORTRAN. We had essentially been told was that IBM was not going to implement an ALGOL processor as an IBM supported language. They said, "If you guys want to do something, build your own compiler with blessings. We'll even help you." Julian Green

and Rex Franciotti were two guys that I interfaced with from IBM, who were trying to build certain pieces of the ALGOL compiler for us. I had given up on trying to keep a cooperative effort going. I had worked rather hard building one piece with a guy at Lockheed in Sunnyvale. We would get together occasionally and check out code to get this and that going. And then somebody would turn up and hadn't done their job, so by that time I got sick and tired of the whole thing and said, "To hell with you guys." I thought that the whole business of trying to get a cooperative ALGOL compiler produced was an abortion. But some people stuck by it, particularly people at Oak Ridge and I can't remember where the other guy was from. I guess he was at Rocketdyne and had a certain personal dedication to the whole thing. Marjorie Lisky and he eventually put together an operable ALGOL compiler which she offered to distribute through SHARE. I don't know how many people took her up on that and I don't know if it ever really worked. It took umpteen thousand passes and probably worked reasonably well. It created at least one social upset. As a result of Julian Green and Marjorie Lisky working so closely together on ALGOL, both divorced their spouses and married each other.

MAPSTONE:

Well, that means they worked well together.

BERNSTEIN:

Yes.

MAPSTONE:

What about UNCOL you said we should come back to that.

BERNSTEIN:

Well, I have a sign here. I keep this. That issue is dated 2–28–58, and that was the dream of the day. The theory said that for any arbitrary source language you could have a generator that ran on any machine that produced the intermediate language version of that program in the UNCOL universal language. You could then take that [intermediate language version of the program] and do the final translation [that would run on any target machine] for which you had an UNCOL translator. [All you needed was] generators and translators. (SDC has never lost that concept, by the way.) Now, no matter what the source [language] was and no matter on what machine it was originally generated [into UNCOL], I could have the intermediate code translated to run on object machine. That is, the sub-set [of the world's computers] we knew about in those days. Note the two interesting things on the end [of the sign]—the Stretch and the Lark. Those were the machines we knew and loved. Remington Rand had 1105 out. The Burroughs 220 was still running, the 1130A and the 205 and the 650 were still live machines. The name of the game was, you had to have a rich enough intermediate language so that regardless of what capabilities were expressed in the source [language], you could

embody that in it. Now I really only had to write one translator per machine. As new machines rolled off the line, I made a translator from UNCOL to that machine language, and all the programs in the world were immediately transferable as long as they had been compiled or generated into UNCOL. Beautiful theory. One of my favorite sayings is very applicable here: In modern science, every day some beautiful theory is destroyed by cruel, harsh reality.

MAPSTONE:

And this was one.

BERNSTEIN:

This was the case in point.

MAPSTONE:

Which was the cruel, harsh reality?

BERNSTEIN:

There were a great number of insoluble problems for a lack of theory and understanding about how rich that intermediate language had to be. We deluded ourselves for a long, long time looking at simple languages. FORTRAN was our basic [source language] model. 704, 709 assembly language was our model the object code at the other end. We deluded ourselves that, yes, you could go from there to there with hardly any trouble at all. We then got into such basic issues as, what does divide mean? Arithmetically, what does it mean? How do you specify precision? Range? How do you do that universally? How do you guarantee that when you take this source language program, which is written, say, in FORTRAN, and run it on the LARK, and then run it on a 704 and then on a Burroughs 205 that I get the same answers out. That is really all you are interested in. If you are going to get different answers out, you're in trouble. So you have got to guarantee a certain level of competence in the language and in the object machine. You can't quite do that because numerical representations and arithmetic algorithms in the machines, even the implementation of floating point, is different between the 704 and the 709, as some people found out to their horror.

I can't remember whether it was done at Lockheed and RAND in parallel, but just before the 709 was rolled in and their 704 [rolled] out they ran a bunch of large matrices which were either inverted or computed the Eigenvalues for. They ran exactly the same programs with only the I/O modified to do exactly the same thing on the 709. Well, lo and behold, the answers were different. Some of them were significantly different; some were only a little different in low order digits. But that shook a lot of people up. It turned out that the floating point algorithms did something slightly different on the "nine" and truncated or rounded differently. That caused a lot of problems. You know that if you are

going to run into the problem at that level, UNCOL didn't have a hell of a lot of chance. Then there was the problem that STRETCH has forty-four bits of precision in the mantissa. The 704 and the 709 only have twenty-seven. How do we guarantee in any way, shape or form that if I do it on the STRETCH first and go back to the 704 that I can get there from here? Or, if I do it on the 704, and accept the answer, that I am not going to get a different answer out of the STRETCH? There were just a multitude of problems that we kept running into. But we actually built a model of an intermediate language. It looked kind of like a vertical language, if I can go back to our earlier conversation. It was sort of a universal instruction set, at least as universal as we could get. We actually got to the level of multivendor cooperation. I can remember SHARE UNCOL committee meetings when we had people from IBM, from UNIVAC, and from Burroughs. The were all interested and excited by the fact that we had gotten far enough along for them to try some of these things. Another delusion. But it is very interesting because the idea keeps getting reborn. Some people recognize that somebody has been there before and used the term UNCOL and some people don't recognize it. Nor will they accept the idea that somebody has been there before. I remember being in discussions and listening to a great pitch about this intermediate form. And I said, "Oh. Hmm. Oh, UNCOL. We were there and have about a foot of file material on what we did and didn't discover." He said, "no, it isn't that at all." [laughter] I'm solving a different problem."

MAPSTONE:

Don't you think this going around in circles is really happening now quite a bit in the field both in hardware, and certainly in software.

BERNSTEIN:

No. No. I think we are inching forward. We went through a fast growth period. All the easy problems and bright people came together and were solved in a variety of ways. There was a lot of shakedown and fallout and we ended up with some reasonably good solutions. But who knows whether they really were. If you look back through time and ask, "Could we have done better?" I'm sure you could absolutely say, "Yes." Did we develop this as a science? No, it is still a craft. Did we achieve a very high level of our craft? Probably not. I had a discussion the other day with a guy who works here. He is English, and has this non-American view of certain things anyway, particularly about crafts. We were able to draw the analogy that, like furniture making, programming is a craft. But unfortunately, unlike furniture making, we've hardly ever gotten a Chippendale and we certainly never have gotten a Hepplewhite. I think that we were so busy just getting anything done, just to claw forward, that we didn't pay attention to trying to get a well-ordered and structured world around us. Because it really was a craft; it wasn't a science in any way, shape or form. And it just grew.

MAPSTONE:

Yes. Well, to be a science, you sort of need some discipline.

BERNSTEIN:

That's right. And we had none. In fact, it was probably the most undisciplined "scientific" endeavor ever taken on the face of the earth by a great number of people. If getting to the moon had depended upon the same kind of technology, we'd still be on the earth. But we are finally, maybe, coming out of the woods, and that is where I see the hope and the light. People are beginning to understand there is a lack of discipline, and that it may well be a craft. That it can be a craft that can be practiced with discipline. But we still have a basic problem. If you look back in the history of this business (it's rather remarkable to look backwards) and ask, "How long did it take for an idea that was a good idea, a phenomenal idea, to really make its way out into the world?" Some of them have taken ten years; and some of them haven't made it yet; they are still floating around. In fact, there are some aspects of this business where we have taken two steps forward and one step back—maybe even two steps back. Look at the relative sophistication of some of the tools that were built back in the late '50's and early '60's, and realize that we have lost a good many of them like the whole debugging tool capability. When you look at a 360 or a 370, and look at the kind of debugging information you get out of that system just for snapshots or terminal dumps, what do you get? You get a page full of hex and down the side you get a page full of characters. From the 704, and certainly the 7090 I was getting formatted dumps where I could specify that I would get the variable name as it was supplied in the assembler and it knew what was a floating point number, what was a fixed point number or what was a character string, and what was instructions. I could get the formatted instructions out, I could get floating point numbers converted back out to decimal for me, and, if I wanted to see something in octal, I could force it. That kind of tool just doesn't even exist today because the information gets generated and thrown away by the system. Somehow, that technology never got transferred.

The basic problem is, this is a technology that only gets transferred by people. You can't write it down and expect somebody to understand it and implement it. There is no standard notation; there is no standard level of competence, no standard language, no standard notation for anything. If I want to transfer technology, I have got to rebuild it. A guy invents or creates something, and then he forms a small group of people around him who become acolytes and then they become apostles. Then the apostles form cells and they send out more apostles, and eventually, if that works properly, the technology gets transferred to the field. You have this seeding and spreading going on. But if you can't make that happen, you don't get the technology transfer. Now when I look back I can understand it. I didn't understand that ten, fifteen years ago. That was wondrous and upsetting to most of us to realize we had created something, had written it down, and nobody was using it. Perspective is what's needed. But that's still going on. Unfortunately, there are whole areas of technology that aren't getting transferred in this business. And maybe in another twenty years we will get over that problem.

MAPSTONE:

You had also talked about JOVIAL

BERNSTEIN:

Yes. The guy you really want to talk to about that is Jules Schwartz. He works for Computer Sciences. JOVIAL was a variant [of IAL]; in fact, JOVIAL stands for Jules' Own Version of IAL. It was named in Jules' absence, I understand. Jules Schwartz was the leader of a project that was attempting to implement a language on two separate computers to do a very, very large programming job, namely, Strategic Air Command and Control System. It was a contract that SDC had with the Air Force at the time. They decided that assembly languages wasn't going to hack it after their experience with SAGE. But they wanted—and I don't know whether the Air Force insisted, or SDC had proposed—that it be implemented in a higher level language. At the time there was no FORTRAN for that computer and FORTRAN wasn't going to be able to handle the real time needs of a system like that. Since IAL (ALGOL58) had just been invented, they decided that rather than try and reinvent the wheel, there was enough body there to build upon and they were going to add the kinds of facilities that were needed in order to do the real time processing job that was needed. So, starting with IAL as a base, they built some superstructure on it to handle the extra things that they needed to do their job.

The story goes that they were back in Paramus, New Jersey, designing this language, and Jules had to come back to Santa Monica or that the language that was being designed here and he had to go back there to settle some problem. As the leader in the language project he was responsible for the whole thing. They had no name for this language. In Jules' absence somebody decided that since Jules was leading this project and he has been very material in describing the necessary additions and changes to the language, they decided to call it Jules' Own Version of IAL.

SDC was for quite some time the sole proprietor of JOVIAL because other people just weren't interested. It is no longer the exclusive domain of SDC. And it probably is, outside of the IBM world, one of the more dominant languages, particularly in the Air Force and, possibly, the Navy. But its users are primarily military, not the commercial and scientific sectors of the world, simply because compilers that do exist on IBM machines are not really supported in the fashion which IBM supports their languages and therefore are not truly acceptable to end users and customers. It also has some properties about it which I am sure people don't like. The current version is J5.5, but I can't keep track of all of the variants they have gone through. Every time it gets reimplemented on another machine, somebody changes something; takes something away or adds something.

MAPSTONE:

Are there any other languages that you are, shall we say, intimate with?

BERNSTEIN:

Well!

MAPSTONE:

[laughter]

BERNSTEIN:

We were having a discussion at RAND about interactive computing back in the dim, dark days when there wasn't very much of that and the guy who was chairing the session turned to Hugh Kelly and said, "You've had relations with the machine, Hugh, could you tell us something about it …" Fortunately Hugh had a very nice sense of humor and said, "Don't tell my wife." [laughter]

Languages I've Known and Loved. I don't know. I have been sort of an assembly language nut for most of the things I wanted to get done. If you are building a complier, you can build a new tool and use it is an intermediate or complier type language. That never turned me on. And so, I have always attempted to work with the basic tools provided by the machine in order to get thing I wanted done completed.

About 1961–62, I gave up the whole language kick as it had too many frustrations and not enough psychic rewards and because I got interested in something totally different. In some ways it related to the language world, but in many ways not. I got interested in interactive graphics, and the kind of languages I foresaw and am still trying to foist onto the world are languages where you use natural notation. If you are going to describe something which is mathematically based, then use mathematics. And we have been able to do that. If you really want to talk about things that are best talked about as flow charts then draw flow charts. That presents a different class of problem, and compiler issues are secondary. My aim and goal for a long time has been to try and drive the interface closer to the end user rather than try to educate people. They [the users] are not in the least interested in the problems involved in the interface.

Modern technology, modern research is a totally different atmosphere from the way the world was ten or twelve years ago with all the problems you run into for funding, and support and who likes it and who doesn't like it. Ten or twelve years ago you got a bee in your bonnet and nobody really cared if you went off on your own time or on Saturdays and you stole a few hours from the company and had fun playing games because you were interested in it. When research became formalized, some of the joy and the thrill went out of it, for two reasons. One: somebody had to be assigned to do it. That closed the door on all the people who didn't have the courage to stand up and say, "I want to do that." And maybe you already had too many people saying, "I want to do that." So you had to make a selection. You ended up with things like sponsors. The sponsor now had a set of goals which may not be in complete concert with your goals. So now you had a little conflict. "Will I do things because I want to do them, or will I do them to satisfy the guy who really is the customer?" There really is no atmosphere of true, honest, basic research.

ARPA funds research and they don't put many constraints on the people whom they fund. They are rather too free to wander but there are still some fences out there. But there is no basic research in this business, particularly with the languages, if you will, or some of the other areas, because we are dealing with artifacts. I'm dealing with the artifacts that I created, and what can I learn that is basic about those artifacts if, indeed, I can go and change the artifacts. Whatever I learned that was basic today is different tomorrow. So there is nothing really basic. The thing is built on a very, very funny foundation. There was a guy at a SHARE meeting who was a very sophisticated programmer. Not an old-timer, but truly an artist; an artist in the sense that that is art on the wall, made a very interesting point. He said, "You know, I got so involved in the programming, I ended up doing nothing." I said, "I don't understand what you mean—doing nothing." He said, "I wasn't doing what I set out to do. I was writing programs, and beautifying programs, and writing more programs, and I got so bound up in that world of nonproductive work—as far as my goal was concerned—that one day I had to sit back and say to myself, "Stop playing with that lovely toy. Stop polishing those lovely stones that only exist in your imagination, and get on with your original goal of building something." And that is what we tend to get trapped in this business.

MAPSTONE:

Yes. That is a very good point.

I would like to take a look at some of the software technologies that appeared on the scene and discuss their significance and importance. Let's start with assemblers.

BERNSTEIN:

Obviously, the first programs were pretty crudely written. In fact, even by the time I arrived in the world, as late as 1954, there was still a certain need for honest to God, absolute binary coding, where there were no aids to help you. And those were the fun times. When you wanted to write a bootstrap for the 701 or 704, or even the 709, you had to do it the hard way. Of course, you had the knowledge and the tools that allowed you to create the thing conceptually in an assembly language. But, when push came to shove, you had to do all the translation, and you ended up hand-punching the first binary version of the thing. There was no way to get it out of an assembler because [assemblers] didn't understand how to produce self-loading bootstrap cards. At least, the earlier assemblers certainly didn't. They all wanted to put forth a card that had an origin, that had a check sum and all the other garbage that went into the first row of a [binary] card, which you couldn't tolerate if you were trying to build a bootstrap. And so, that kind of craft was certainly extant through about 1956 at least, probably 1957. I think it has been long lost. I don't think there is anybody around, may be a few, who came into the business after 1960 who really has ever sat down and created a binary program, which was the way things literally first got into a machine. There may be a few people around the world, inside IBM, who were building test machines. It is certainly not something anybody does as a regular practice today.

The first assemblers were an attempt to get over some of the bookkeeping. The subroutine philosophy, rather than running large, monolithic programs, was already in existence in 1954. I don't remember being told about subroutines in 1952 in the class I took from Cannon and others. I have no recollection of them talking about subroutine in those days. By the time I got to RAND, subroutine and the concept of presenting versus post setting was a hot issue; a very amusing hot issue. Philosophically, there were two ways about using a sub-routine. You passed the location in the calling program to the subroutine in the accumulator. Remember, we had no index registers on the early machines. You set up all the addresses to refer to the data that was necessary to pass to the subroutines by taking that initial linkage address out of the accumulator and planting it in address fields appropriately modified to grab the data that was immediately in the vicinity of the calling sequence. Then you also set the return [to the calling program]. At that same time you had any initialization to do, like you had set certain parameters to zero or an initial value, you did that. You then executed the subroutine and you left it in whatever condition things were sitting in when it came time to exit either by virtue of discovering an error or getting the answer you wanted and depositing it where it was supposed to go and going about your business.

But there was a whole class of people who said, "No, no! you always leave sub-routine in its pristine condition, so that when you finish, you post set it. You reinitialize all those variables so that the next call doesn't have to do that and blank out all those addresses [of the data passed by the calling program], so that when you exit it looks like it was when entered." I think the presenting philosophy began to dominate, at RAND, at least. I don't know what the rest of the world was doing. That wasn't a burning issue outside [RAND] but it may have been. You know, there was a great stir about that for a long time. It was finally agreed that presetting was the way to go. That was the only way to guarantee that what you needed was going to be there when you needed it. So the first assemblers attempted to give you the facility of creating the subroutines in blocks that could be address independent—or at least origin independent at the assembler level. But they certainly were sequence-dependent because they were relative assemblers, not symbolic assemblers. One either picked a letter of the alphabet or a two-digit number and put a period between it and the sequence numbers, depending on which assembler one was using, to identify the block. Everything [in that block of code] had to follow in exact sequence after that. If you had an insert to make—like you left a line of code out—you now had to renumber everything and recode all the references because all the references were relative to the origin, but absolute within the block. And so, you were really back in the octal or decimal absolute coding business. The only first semi-symbolic assembler was written at Douglas [Aircraft] that allowed you to insert and delete in sequence without high penalty. Although you still had to reassemble, it was a small price to pay because the assemblers were respectably fast.

I have no idea who dreamed up the first symbolic assembler. Cliff Shaw [17] built one that was probably running by the time I got to RAND. It had some strong constraints on how you could name things and where the blocks had to be. It certainly wasn't block-structured in any reasonable sense of the word. You just wrote a big program. You could

use subroutines or not, as the case may be. And the subroutine labels were the same as any other labels.[18] The first real, complete, honest to God, symbolic assembler that I can remember was SAP for the 704, although we had a fair symbolic assembler running on the JOHNNIAC, but I can't remember what it was called. [19] There may have been other symbolic assemblers running in the world on other machines. I think that the concept came to fruition with the 704 with the ability to start over from scratch and build one [an assembler] to meet all [of the programmer's] constraints and the needs. That opened up a very, very funny world to people. A lot of old programmers by that time, had been used to using non-mnemonic names to locate things, to identify elements of the code, varying variable names, what have you. The new freedom of naming [things] with six alphanumeric characters of your choice opened up a delightful world. And people invented all kinds of crazy naming schemes for things so that they could keep track of them themselves. Then it became fashionable for a while [to use English words] and in some cases it became dirty word programming or dirty limerick programming. If you read the labels in the word order as they appeared on the listing, you might really end up with a very delightful, plain or dirty limerick.

MAPSTONE:

Do you remember any?

BERNSTEIN:

No. No. I just remember seeing some of those things. And I can remember one guy who was exposed to this freedom to label the branch points that you branch to from decisions with something that could tell without having comments that this is the place you went from decision point A in which the answer was positive or this is the point you went from decision point if it was negative and so forth. He just got completely carried away and almost every line had a symbol. He just got completely carried away and almost every line had a symbol. He just destroyed the utility of the whole idea. He was one of the few people who probably overflowed the symbol table. That art came to reasonable fruition rather quickly. It didn't take long to transfer that technology. That was an obvious one. What took a little longer was the concept of macros. I guess Doug Macilroy of Bell Labs may well be the inventor. He is at least one of the early pioneers in the whole macro business. He may have published a paper in the ACM Journal, or maybe in an obscure Bell Lab journal. I associate his name rather closely with macros. He may still be at Bell Labs, by the way.

MAPSTONE:

In New Jersey?

BERNSTEIN:

Yes. But there are three of them there. I can't tell you which one. I don't know whether it is Whippany, Murray Hill, or New Brunswick.

MAPSTONE:

Or the other one.

BERNSTEIN:

Yes. New Brunswick or something like that.

MAPSTONE:

That is MacIlroy.

BERNSTEIN:

Macros went through a very funny faddish stage. The concept was reasonably well understood, but the methodology—again you are now away from something that is reasonably well understood, like a symbolic assembler which doesn't give you a terrible amount of freedom; there aren't very many ways of playing that particular game. But the macro game could played in a lot of ways. I guess, except now by default, you have the IBM 360 assembly language, and the macros are embodied there. There were probably for the 704, 709, 90 series at least a half a dozen different macro assemblers interfaced either as preprocessors or that were built right into the various assembly languages that were around. And the macros certainly changed between SCAT, which was the assembly language which went with the SHARE Operating System, and the MAP which was the IBM produced assembler for the 7090 which went with IBSYS. And there were several other funny aspects to the assembly technology which came about at that time. On the 701 it was very, very difficult, if not impossible to talk about a load and go system.[20] Some of the things we built for the JOHNNIAC which were rather crude and simple, could get a load and go facility. But storage was always a fun problem. I don't remember any load and go facilities on any of the assemblers that existed on 701 [see the above end note]. We eventually had an assembler written by Cliff Shaw again for the JOHNNIAC that was a load and go system. But you paid some penalties. You had to structure your program in a particular way which allowed the assembler to take advantage of you as opposed to vice versa.

It wasn't until the SCAT assembler or the SHARE Operating System that the load and go philosophy reached some kind of nadir or peak. The concept was to take symbolic source language and pass it through the SCAT assembler. It produced as its first output something called a "squoze" deck which was all the information contained in your source, but compacted down very tightly. There were some very interesting compaction schemes generated. You could further declare that you wanted a binary deck produced. But there was really hardly any reason to take the binary deck. What you really wanted

was to execute whatever the output of the assembler was [immediately]. You could force execution even if there were syntactic errors discovered in the assembler. So you had a way of taking symbolic programs, throwing it in [to the card reader] and getting a run made. You could either take just the results of the run as output or you could take the results and a squoze deck and/or a binary deck. There were a number of options. The philosophy was, once you got your squoze deck, you could modify the squoze deck with mod cards that went on the back of the squoze deck. You certainly couldn't read that [squoze] deck very easily. You had to be a minor magician to do so. What they had done was to take [the results of] the first pass of the assembler and put it all together so that you only needed a second pass of the assembler to get the program to run.

MAPSTONE:

I take it squoze comes from squeezed.

BERNSTEIN:

Yes. Well, it was the past pluperfect possessive of whatever was squeezed. I could reconstruct how it may have come about. Do you remember the old joke about the Boston cab driver and [the passenger who wanted to eat] scrod?

MAPSTONE:

Yes.

BERNSTEIN:

The past pluperfect subjunctive - yes.

The object of the game was to take the squoze deck and either run it through the second pass of the assembler only, which is very swift, but more importantly I could make changes to that deck without ever having to go back to the symbolic source again. I don't have to load the big deck again. I load this very tightly compacted deck and I only run it through the second pass of the assembler. Unfortunately the implementation left something to be desired. Scat—the first pass of the assembler that produced the squoze deck—had its own second pass instead of using Modify and Load, which accepted squoze decks as input. There were some minor differences between the two [second passes]. It caused a few very strange problems. SCAT had a bigger macro tables than Modify and Load. You could get a perfectly fine 7090 run from your first symbolic pass at the machine, and when you would come back for a second run later and ran [the squoze deck] through Modify and Load, you got aborted because you overflowed the macro tables. Or you got different numerical results because the number conversions in Modify and Load were different from the number conversions for SCAT. There was a bug in Modify and Load. That always shook people up.

FORTRAN was a six phase compiler and I can't remember whether it was phase 5, or phase 6 that did index register assignment. It literally went through the whole program and laid out a tree of the flow. It saw what was needed where, climb around that tree, and make index register assignments. Almost completely independently of that invention by Backus and company back in Poughkeepsie, the people who did PACT did exactly the same thing and I am pretty sure Tom Steel and somebody else were responsible for that. They invented the same technique for assigning index registers on (or indices) into the program. It was very strange. Having dug into that piece of FORTRAN at one time and discovered later having also dug into PACT that here was the same algorithm independently invented and independently put together by two completely different sets of people to solve exactly the same problem. It was not a terribly obvious solution. I'm sure that the people strained, struggled and groaned an awful lot before they came up with that solution. It was not one that, if you handed the problem to somebody today, would not be forthcoming without an awful lot of pain and struggle. But it may be the only solution after you think about this very logical thing that you have to do. It was a rather amazing development that happened almost simultaneously.

MAPSTONE:

That happen sometimes today in science or medicine or something, where two people, on different sides of the world who don't even know each other invent the same thing.

BERNSTEIN:

Well, I'm sure all the cast of characters knew each other, but I don't think they were in close communication. In fact, there may well have been a certain feelings of competition between the two groups. Again, the vertical and the horizontal types were at least verbally battling each other.

I don't think that other than the invention of FORTRAN or COBOL, the whole problem oriented language development, if you will, or the assemblers, the macros, the operating systems has had any real impact on the art and craft of programming. They were touted to be magic that would solve all it kinds of classic problems that the world was faced with. And yet none of those tools and none of that technology addressed some of the basic issues. The basic issue, at least from my point of view, has always been how organize a problem in order to get it done in some reasonably and orderly fashion before you put the pencil to coding pad. That is where the craft, the skill and the art come in. Some people practice it at a very high level and some people practice it on a very disorderly and low level. None of these languages has helped to do anything about that. FORTRAN may have been guilty for introducing main frame CPU inefficiency beyond all reason. Of course, they got the job done rather quickly. What they did do was allow people to get a job done more swiftly, more efficiently from the point of view of trading off people time against letting the computer do that which it could do very well and preventing people from constantly making dumb errors. But it didn't do anything for making people think more logically and more orderly.

MAPSTONE:

Are you saying that as yet there really has been no tool discovered that has had an impact on the quality of programming?

BERNSTEIN:

That's right. I think the quality of programming is still a very personal issue. We may be in a world where people are touting magic again. Maybe structured programming or chief programmer teams or whatever you want to call that kick that we are off on now, will bring some order out of the chaos. But I think the order will only come by education, not because there is a magic tool there. The benefit will come because we are beginning to teach people about program organization as opposed to the programming languages.

Programming languages, per se, have no organizational ability built into them. In fact, no machine level language, I don't care whether it is a macro assembler or a octal assembler or what have you, tells you anything about how things should be organized. Some of the higher level languages impose a minor structure on you in that you can see some obvious things that you should and shouldn't do, but they still don't tell you what's a module and what's sub-routine. That is an individual or independent decision, or a group decision if it is a large program. The organizational structure of the program is what determines its overall quality, maintainability, usability and reusability. And there is nothing inherent in any of the stuff that has gone on to date that solves that problem. Nothing tells you what orderly approach to use to solve the problem. So what we have done is given people sharper pencils, in a sense, and nicer coding pads, but we haven't done anything to the organizational requirements they really face. And we don't teach that to anybody, either.

MAPSTONE:

Presuming one starts with a programmer who can think logically and clearly, some of these devices, assemblers, compilers, interpreters, etc., have made it a faster process?

BERNSTEIN:

Somehow they have made it simpler and easier to get the embodiment of the program onto paper and into the machine more quickly. Interpreters are a funny side issue. Interpreters allow one to specify very complex processes in great simplicity, if they are done right. There were some delightful programs that were really interpreters for providing the floating point on the early machines that had no floating point hardware. That was one very valuable problem [to solve]. You don't want to have to call a subroutine with some funny linkage when you could write an interpretive program. You paid a price [in performance], but you could certainly think about floating point arithmetic much more directly in a language that was analogous to whatever assembly language you were using. You give a special call to the interpretive portion of the code

and it looks very much like the other code that you had been writing. Then you give another special call and you are back in normal assembly language.

There were some interpreters that were full packages like the Douglas Matrix Abstraction. The package that Douglas built was an interpreter that handled matrices for you. If you wanted to multiply two matrices together, early in the program, you defined the size of each matrix and named them A, B, C, D, E, and said multiply AB and stuff the result in C. That allowed you to handle complex structures with the same simplicity that you would write an ordinary arithmetic routine. But you had to be careful. You had to understand something more than simple arithmetic. You had to understand what you were doing when you transposed a matrix and what you were really doing when you tried to invert it. You knew you were chewing up a lot of machine time in what looked like very simple one line operations. They had a very powerful effect in showing people that there were simple ways of expressing their problems, but also they put limits on you. If you didn't like the way the man who wrote your matrix package did matrix inversion you were out of luck. If you didn't like Gauss-Jordan or your matrix was ill conditioned then you had to find some other way that to invert it. If you didn't like the way [the package] did floating point arithmetic or double precision arithmetic, which is in the class of things that interpreters were good for [you were on your own].

As [higher level] programming languages came along [interpreters] were a much more rapid way of implementing a higher level language than building a full compiler. In fact, I and several other people [at RAND] built a couple of interpreters. They were fun. They were easy. You could start with a standard basic package of goodies an design a very quick and dirty language and try it. You could quickly modify it because you didn't have to build a back end which compiled into full executable binary code. But you pay a penalty if you do that. The penalty is that it runs a hell of a lot slower than you ever want it to if you were going to do productive work with it. Part of the problem was that we tended to delude ourselves that the beauty of the front end was a perfectly reasonable sacrifice to make for the efficiency of that back end. That is why a lot of stuff dies. But, again, noting in this world helps to organize your problem. They are all pretty languages, and you get to the machine more quickly.

Time sharing came along and spurred a lot of developments in the language and the interpreter area and in the assembler area. But in the actual program or production aspects of the world, what time sharing allowed you to do, more so than anything else, was rapid update if you had a proper file system for your source code, and swift debugging. But it didn't help the conceptualization of the problem at all. In fact, studies proved that (1) they are very expensive, (2) they are hard to control, and (3) the results are very difficult to interpret. One interpretation of a major study done here [at SDC] by Hal Sackman says that the between people variation is an order of magnitude bigger than the between systems variations in the trade off between batch and time sharing. Give me a good programmer. That is what I really want if you want the job done right and quickly. Give me the best programmer you have. No matter what system and tools he has to work with, he will get the job done better and faster than any poor programmer, no matter what tools

you supply. And so the people variation is still the dominant one. All the tools in the world aren't going to help that one bit until we educate people properly. Maybe structured programming is the answer. Maybe we finally discovered the organizational problem is the main one. One of the things that I thought would (and it probably is a delusion) be helpful is to allow people to write programs in the form of flow charts and never have to worry about any form other than that. If I think about that for a while I can just as badly disorganize a program at the flow chart level as I can at the FORTRAN or COBOL level.

MAPSTONE:

What I remember of my programming course, that was where you really got hurt the most.

BERNSTEIN:

Yes.

MAPSTONE:

The philosophy was that if you haven't got it there, you are never going to get it down on paper.

BERNSTEIN:

But there is at least fifty percent of the world population [of programmers] that doesn't believe in flow charts except as documentation after the fact. Be that true or not.

MAPSTONE:

It is not what IBM is teaching,.

BERNSTEIN:

I heard that IBM was no longer using flow charts. They are using something called HIPO. I don't know what it is.

MAPSTONE:

Oh. It sounds like something one uses in photography.

BERNSTEIN:

Yes, I know, that is what it struck me as.

MAPSTONE:

It smells awful, right? I took a programming course in February a year ago, and at that time we spent weeks and weeks and weeks on flow charts. And really you got your wrists slapped time and time again if the flow chart was wrong.

BERNSTEIN:

Well, I just heard Ted Climus say last Monday or Tuesday that IBM wasn't using flow charts. They were using something called HIPO, which had been invented by the tech writers and which the programmers love because it was better than flow charts. I want to see that.

MAPSTONE:

Sounds interesting.

BERNSTEIN:

Yes. I'm going to go chase that one down.

MAPSTONE:

Excluding the ability of the programmer to use his brain, what would you say were the most significant developments in software technology? Is that an answerable question?

BERNSTEIN:

It is very, very difficult. One of the landmarks is obviously FORTRAN. Just the conceptualization that there was a better language tool for the world if you could only solve some problems. It took a long time to solve all of the problems, and I think that the compiling and the language technology today is in pretty good hands. While all these design problems have not gone away, that is a very personal thing again. It is like what style would you like to write in, what suits your personal predilection. Do you want to drive a General Motors or a Ford automobile, do you like APL or do you like PL-1? That is based upon a hell of a lot of preconditioning. I think there is a basic pattern that flows through this whole business: That which you were exposed to first is somehow the thing you love the most. I don't care whether it was 701 assembly or whatever. There are still a lot of people running around saying. "If we could only go back to the 701." And they only say that half facetiously.

MAPSTONE:

Yes, that is right. There are people who say, "If we could only go back to the CPC.

BERNSTEIN:

I don't know anybody who really wants to live with that, and all of the ugly problems that it presents in the world.

MAPSTONE:

Some of its concepts, then.

BERNSTEIN:

Yes.

I don't think symbolic assemblers really are a landmark that stand out clearly. I think they were an evolutionary phase where people began to learn that the machine was good for something other than straight numerical operation. That it has the power to do something other than number crunching. Therefore, we evolved through the set of assemblers we had to some kind of reasonably respectable assembly programs with macros and other kinds of aids. Was the concept of local versus global variables and symbols a landmark in the world? Who invented it? I haven't any idea. It may have occurred to a thousand people simultaneously. Does it really buy you something? Not clear. Again, it doesn't help to organize things any better. It just frees you up from some of the mistakes you can make by calling three different things X so that when you try and assemble all the blocks of the program you discover that you have ambiguous symbols. Wonderful. Those I believe are strictly evolutionary trends that would have happened no matter what else happened.

FORTRAN stands out as a landmark from one guy who has to be patted on the head. It was out of the mainstream. Are there any others that qualify in that area? I don't know. PL-1 was a compromise. They [the higher level programming languages] are all variants on the same theme. Someone invents a new stream of technology and everybody climbs on board. Then you get an evolutionary movement on that trend. Operating systems was move towards rational efficiency. It was also a move that stifled a lot of things. Operating systems impose accounting on people. You have got to have a legitimate job number and identification in order to get the job accepted by the system. But the removal, in the better operating systems, of the nitty-gritty in performing the I/O and the isolation of the I/O and the data management functions is certainly a major help in getting the job done. Conceptually, where do you start with the first operating system? Do you go back to General Motors where they probably used the first one on the 704 back about 1957. I think Bob Patrick could clear that up and give you the proper dates. He was there at the time. I think that comes close to being the first workable operating system. There was an evolution through SCAT then IBSYS and now OS[/360], but I don't think IBM is in the mainstream. I think IBM is following on the previous wave of people in the mainstream.

There are operating systems which are certainly superior in many ways to OS [/360]. But the concept that you could remove a lot of the nitty gritty from the programmer, not the programming or the housekeeping, but the I/O management and systems resources

management is a concept which obviously is an aid to getting the job done swiftly, but not necessarily better. Now, it does do some things better. It precludes making stupid I/O errors and stupid resource management errors. But it also imposes an overhead on it. So there is a price you pay, and the price you pay is there because they try to build [an operating system] that satisfies the needs of an entire class of users you believe is going to have to use the system. Now you get into the problem of what is the proper tradeoff. What [features] should I provide? How sophisticated a functional capability should be in an operating system? But I think that, again, it is another standout item that follows a stream of more and more capability and realization of power as it comes down the pipe. I think those are the only two real landmarks that stand out in my mind.

The computer itself is the bread and butter item. [Programming] languages is another bread and butter item because you can show economic advantage just as the operating system shows economic advantage. Time sharing is a variant. It has certain advantages. It certainly is a landmark in the world but it comes outside this time frame you are talking about.

MAPSTONE:

RAND was an unusual place. When you were there, did you get any feelings during the 1950's what the true significance of the computer was? How early did people start getting into social significance and problems?

BERNSTEIN:

Not in the '50's. In fact, I can remember one incident where Edmund Berkely stood up and gave a paper in a meeting, and said, "One must beware of the social consequences of this magnificent tool which can turn into a Frankenstein," or some such words. And some of the people from RAND who were there came back and said, "That guy is a nut. He doesn't know what he's talking about. What is this social significance garbage? You know it is just another tool. There is no menace in the tool." And it took another three or four years, as far as I can remember, that people began to realize the real impact of this tool. You see we were living in a funny world at RAND. It was the world of the scientific application. RAND, like the cobbler's children, was way behind in using the computer as a tool in the management of the corporation. It wasn't until the 704 had been in-house for quite a while that a payroll system was written for it; which recalls a very funny incident about not leaving enough space for the number of dependents, and blowing the program one night, much to the consternation of the gal who was supposed to run the payroll. She couldn't get it to go. And the guy who wrote the program wasn't cleared to look at the sensitive salary information and so there was a basic problem of how to find the bug. How do we recreate the bug when we don't know what data caused it? They finally over that one.

But RAND wasn't in a production environment in the sense that we weren't building airplanes. We [the Computer Science Department] were satisfying needs of a set of

customers, the other RAND technical and scientific staff who were trying to solve a different class of problems—very complex models. I can remember spending about six months working with some people building a huge war game; the air battle type. It was primarily a nuclear war, therefore it was fought all in the air with bombs dropped. But it was horribly and deeply complex. But no social implications were involved at all. We were in the vanguard at RAND with many of the uses of the computer in exotic areas. RAND was one of the two places where artificial intelligence got started. Whether you believe in it or not is irrelevant and immaterial. Newell, Shaw and Simon started one track, Minsky and McCarthy, other. Those were the two clearly identifiable sources of using the computer to do something for which the creators and the builders of them had completely failed to anticipate. The whole symbol manipulation and attempted emulation of human thought and human thought processes were nowhere in the minds of the people who built the first two generations of computers until after some of the basic work was done. The fallout from that was probably more interesting than work itself: The whole business of list structures and alternate ways of representing data and content addressable collections of information rather than an address pointer. I could find something by looking for the content, by hypothetical associative store to working down the equivalent name in the list structure to find out where the thing I want is without having to know beforehand its name and being able to point to it directly. That opened up a whole vista of capability which had an impact on compiler-building. It had an impact on all kinds of applications. Whether it is a landmark or not, I don't know. Maybe people who needed it would have invented it independently if it wasn't there already.

MAPSTONE:

It was significant.

BERNSTEIN:

Yes. It was a significant idea. It eventually led people to literally thinking about data structures. But when IPL-5 and LISP were invented, data structures wasn't a phrase in the vocabulary. We had lists and list structures, but they were a very particular things. But it eventually blossomed into something that had impact over a very broad segment. This whole business of data management and data structures had a tie back into the concept of that you don't have to stuff a word in the place of memory where you know its exact location and that is the only way you will really get at it again. I can tie things together in most delightful ways, once you have the concept of the list of pointers and classes of things and trees and nodes and all the good things that go with it.

RAND really took off in some very, very interesting directions and that was one. Is it really appropriate for them to do that? Sure, probably, at the time. The fact is that the whole thing eventually died. Nobody kept the fire going. I don't understand. RAND was a hot bed of potential and somehow somebody managed to turn the fire down under the pot and it stopped boiling. That happened as early as 1958.

MAPSTONE:

One person said they felt it happened when John Williams died.

BERNSTEIN:

No, it happened long before that. It happened no later than 1959.

MAPSTONE:

When did he die?

BERNSTEIN:

Oh, he died after I left, and that was 1963.

MAPSTONE:

I understand that he was a great catalyst, and was also interface between upper management and the scientists.

BERNSTEIN:

Yes, he had some very positive, strong effect to about 1956, '57. My best recollection says that John Williams was literally out of the effectiveness loop of computer science effort by then. I did a job on the JOHNNIAC for John (it was about 1956 or thereabout) a game theory problem. I ran the program in a variety of ways, and discovered that there was a flaw in his theorem. It upset him a little bit. I came in to his office and I said, "Dr. Williams, no matter what I do, the distribution is normal. No matter how I perturb it, I can't get it off a normal distribution. I don't 'think that is what you intended." And he said, Let me see that stuff." And that was the end of it.

MAPSTONE:

How about the reverse of everybody clutching at the machine? What about the negative reaction. Was there any of that? Were there people, especially mathematician-scientists who were saying, "No, no, this is not the way to go."

BERNSTEIN:

Yes. Yes. In 1956 or thereabouts, a man [21] whose name I can't remember came to the department and said, "I want a program that will give me some lunar trajectory information." And we said, "Oh, there must be some somewhere in the world that we can get the numbers for you." We discovered that there wasn't anyplace that we could lay our hand on the data. Nobody had written [computed] anything other than an elliptical

intercept between the earth and the moon, which is not a true, proper solution to the three body problem. So Nancy Brooks and I were given the assignment to program an integration program that would integrate the differential equations representing the earth, the moon and a vehicle in a simple two dimensional circular orbit [for the moon]. It did not include an ephemera for the outer planets nor was the gravity of the sun taken into account. It was a simple three body system. [One important condition of the effort was that the vehicle was to arrive at the point of equal gravitational potential with a minimum of velocity toward the moon because the vehicle was going to crash on the lunar surface, not land. Minimum velocity at the point of equal potential guaranteed that the vehicle would impact the lunar surface at the lowest possible velocity to assure the survival of the instrument package it carried. Such a lunar mission was actually carried out a few years later.] We wrote a program using Runga-Kutta integration because that looked like it was very appropriate for this problem. We validated the program by making sure that the vehicle and the [22] moon stayed in their respective orbits for twice as long as the expected five day transit time to the moon. In addition, we tested the stability of the integration by placing the vehicle at one of the earth-moon libration points to verify that it would remain there for at least ten days. The program passed both tests perfectly.

Using the initial conditions (point of insertion relative to the earth-moon axis, insertion angle and velocity) for a flight to the moon provided by one of the engineers who worked for George Clement, we proceeded to compute our first lunar trajectory. It was a miserable failure. We missed the moon by thousands of miles. We concluded that we had been given a poor guess and modified the inputs. After a number of unsuccessful tries, we managed to get the vehicle to hit the side of the moon facing the earth, but the insertion velocity was twice that of the original data we were given and the vehicle crashed into the moon at some horrendous velocity guaranteeing total destruction. We were quite confused and upset, but too proud to admit that there might be a problem with our program.

Shortly before we started the program, RAND had installed a large flat bed plotter on the JOHNNIAC, the machine we were using. In desperation, we decided to plot the positions of the vehicle and the moon on the plotter instead of printing out tables of numbers representing the trajectory. The first run after including the plotter was most enlightening. Within a few cycles of the program, the problem of not being able to arrive at the moon at a reasonable velocity became apparent. The original parameters for the equations representing the earth, moon and vehicle had given us a retrograde moon. The earth was rotating to the east and the moon in our simple system was orbiting to the west. One small change, and we were in business. We were able to refine the original conditions given to us so that the vehicle would arrive at the point of equal potential at about one foot per second velocity toward the moon and impact on the side of the moon facing the earth. All of this was done in a matter of days.

The results were given to the Engineering Department and we assumed that we were through. A few days later, one of the engineers came to my office to discuss our results. He was sure that the computer was wrong. He had spent the several days between the

time we gave our results to his department and the time he came to my office laboriously hand computing the first several hours of the trajectory. His concerns were that ther error of the integration scheme we had chosen was too large because our five digit results were "significantly" different from his hand computed ten digit results. Therefore, the computer was not to be trusted. I pointed out that the error function for Runga-Kutta-Gill was of the order of integration interval h squared, which in this case was about 4 times ten to the minus 6 because our interval was one over 512. This was sufficiently small to assure that our results were as accurate as one needed for the intended purpose. I then asked why he had used ten digit precision. His response was that he always computed things to at least ten digits. He wanted to know why we had only used five. I pointed out that the computer used 39 bits which was equivalent to ten digits or more, but that we only printed out the five most significant digits for good reason. Namely, our computations weren't accurate beyond two digits. He took umbrage at that saying that I was implying that the data that the engineering department had given us was inaccurate. I then asked him what he used for the mass of the moon relative to that of the earth. He replied that he used the same one we were given, one over eighty-four. I pointed out that neither of us had more than two digits of precision and that as long as his and our calculations agreed to two or more digits, there was nothing wrong with our program or the computer. He never brought up the subject again, but I was sure that he thought that the computer was a device not to be trusted.

[End of Interview]

[Endnotes]

[1] The project failed not only because they tried to chew off too big a part of the Army to model, but because they managed to upset a large number of senior officers (colonels and up, including one four star general) when they invited about 20 of the top brass of the Army to Philadelphia at UNIVAC to see the model successfully demonstrate how it could minimize the cost of putting an army in the field. The person in charge of the project had talked UNIVAC into providing the machine time needed to check out and test the program and run the demonstration on the promise that, if successful, the Army would buy a machine. What they forgot was that for a problem the size that they had chosen, the program would run for many hours. After all of the speeches telling the collected brass how great a breakthrough this was going to be for budgeting the Army, once the computer started running the program, there was nothing to see or do. After about an hour of standing around with nothing to do, the four star general departed as did everyone else in a very sour mood. I never found out whether or not the program ran to completion.

[2] Standards Eastern Automatic Calculator. I don't think that the word computer was in use for machines by that time. Computers were people who performed computation using pencil and paper and desk calculators, like Fridens, Monroes, and Marchants.

[3] Standards Western Automatic Calculator installed at UCLA's Institute for Numerical Analysis.

[4] NSA actually got 701 number 4. The first 701 went to IBM World Headquarters at 590 Madison Avenue in New York City, the second went to Los Alamos, and the third to Lockheed Aircraft in Glendale, California.

[5] Actually, I only worked for Atlantic Research for about 20 months from February 1953 until October 1954.

[6] I recently (May 1996) had occasion to revisit that one card bootstrap loader in conjunction with a personal project related to the history of computing in the early days of RAND. It was almost as inscrutable now as it was then and without my old 701 manual, I never would have figured out how it worked.

[7] When I got to RAND in 1954 I was told that as a member of the Technical Staff I could spend up to 20% of my time on work of my choosing. By the time I left in 1963, that was no longer the case.

[8] I believe that the actual rental cost of the IBM 701 was something like $60 per hour for three shifts a day, which only comes to $1 per minute. Perhaps my memory was off by a factor of 10 or management didn't dissuade us of our error in order to discourage wasting time.

[9] Ms. Mapstone found a reference in: Dantzig, George B., Linear Programming and Extensions, The RAND Corporation, Chapter 27, Stigler's Nutrition Model: An Example of Formulation and Solution, pp 551–567.

[10] Remember, the 701 had no index registers. All address modification was done by modifying the program. Thus indexing in PACT I was a major step forward.

[11] There really were only three index registers on the 704, numbered 1, 2, and 4. This allowed the computer to perform multiple indexing without doing a lot of extra arithmetic. Thus if an instruction's indicated index register was 5, the contents of index register 1 and index register 4 were ORed together. If one specified arrays whose dimensions were powers of two, one could enjoy the benefits of multiple indexing in a single instruction.

[12] Apparently, the idea for COMMON in SAP came from the early assemblers for the IBM 701 and other computers that preceded the IBM 704. These were what was called "Regional Assemblers." the earliest of these were not symbolic assemblers (though some of the latter ones included some symbolic capabilities), which made making insertions into the code quite inconvenient. The subroutine libraries that accompanied these assemblers used several standard "regions" whose origins would be specified by the programmer in his program. One of these regions was used for temporary or scratch

storage and was the same for every library routine. Thus the programmer only had to determine the largest one needed and set its size accordingly.

[13] According to the Proceedings of the History of Programming Languages held in 1978, FORTRAN III was only ever used by IBM internally.

[14] The four U.S. delegates to the first International Algebraic Language meeting held in Zurich, Switzerland in 1958 were John Backus, Charles Katz, Al Perlis and Joe Wegstein.

[15] It was a letter from GAMM, a German technical organization, to John Carr III who was president of the ACM at the time, that started things off.

[16] Apparently, the name ALGOL had been agreed upon at or shortly after the Zurich meeting, but was not publicly acknowledged until sometime later.

[17] In looking through some "ancient" RAND program library write-ups, Cliff's assembler for the IBM 701, JCS-13, wasn't completed until April 1955, some months after I arrived at RAND. It was not, strictly speaking, a symbolic assembler. It was a Relative/Symbolic Assembler. It did provide relief from the tyranny of the rigid sequencing of relative assembler. But if you wanted flexibility in organizing your code, you had to write the program in blocks. Within a block, you could label any location with a symbol (rigidly constrained) and were free to insert an delete lines of code at will.

[18] Not really. To quote from the JCS-13 write-up dated 4–26–1955, "Symbolic locations and addresses are of the MAPSTONE: blank, numeric, numeric, alphabetic (zeros need not be punched). … Regional locations and addresses are of the MAPSTONE: one alphabetic character for the region symbol followed by a three decimal digit sequence number (zeros need not be punched except in the units position)." Thus there were 2,600 symbols and 26 regions, but a region could be reassigned a different origin at the programmer's discretion. To quote further from the write-up, "In coding instructions, the programmer should use symbolic locations in order to take advantage of the insertion and deletion facility. … In setting up data storage the programmer should use regional locations to minimize the number of symbols he has to define. For example: E 0 SK 20 will skip 20 half words in assigning storage to the program. … To set up 20 half words with symbolic locations would require the programmer to write out 20 such instructions to define the symbols properly."

[19] It was written by Jules Schwartz and was called "J100A JOHNNIAC Symbolic-Relative Assembly No. 1" and was very similar to Cliff Shaw's JCS-13 for the IBM 701.

[20] Cliff Shaw's JCS-13 actually did operate as a Load and Go assembler.

[21] The man was George Clement who, at the time of the interview, was at the Aerospace Corporation.

[22] The tape ends here, but at the urging of Ms. Mapstone, I've finished the anecdote.