METAPRINT 3

(METAMETAPRINT 1)

Responses to "COMPUTERIZED LINGUISTICS: HALF A COMMENTARY"

- Martin Minow -

Rather than attempt a summary of the replies to "metaprint" 1 included here, I feel it would be more useful for me to discuss one of my programs.

PURPOSE

The program generates sentences from a generative (context-sensitive, transformational) grammar. It is almost identical in function to that presented by Joyce Friedman in preprint 14.

LANGUAGE

While I had previously written a context-free generator in assembly language, these programs were written in SNOBOL3, which is intended specifically for string processing. There were both advantages and disadvantages inherent in this choice. The language provides simple, powerful operations for parsing strings and allows easy defination of push-down stacks and lists. In addition, a primitive was available which recognizes strings balances with respect to parentheses. Because of this, I chose to represent trees as fully parenthesized strings. What is more important than this is the fact that SNOBOL manages all storage automatically. Thus the program has almost no pre-defined limits.

The major disadvantage of my choice was that I was completely inexperienced in the language and unfamiliar with the recursive techniques permitted. Thus the program was extraordinarily inefficient.

THE PROGRAM

Bot the context-sensitive generator and the transformations program were written in two separate stepts, one converting the rules from a form as similar to that used by the linguist as possible to a form convenient for storing on the machine. In general, all rules containing abbreviatory devices (braces and parentheses) were expanded to a number of sub-rules. These were then punched out and used as input to the generator itself. The size of the programs is

CF generator (assembly)	2500	cards
CS rule reader	300	statements
CS generator	~ 360	
Transformations rule reader	~ 280	
Transformations executor	~ 600	

In addition, each program contained approximately one comment for each 4 statements since this was the only way I could understand what I really intended to write. These were written as the program was written and proved invaluable. I should note that I generally over-document my programs as I tend to borrow algorithms from them years later.

DEVELOPMENT TIMES

The CS program took about six weeks to get running while the transformations program took the better part of four months before it worked well enough that I could attempt to transform a "real" sentence. Due to personal reasons, I was unable to debug the grammar/program well enough to consider distribution. The cost of processing a real tree was also prohibitive (20 minutes at 7094 time). I again note that this was caused more by my inexperience with SNOBOL than by any faults inherent in the language. The work could hardly be done so quickly in assembly language or FORTRAN as I first would have to write a large set of subroutines for string handling, input-output, etc.

INFLUENCE OF THE LANGUAGE

The strongest external influences on the program was the fact that data must be punched on cards. Thus a two dimensional notation, as used by Fromkin and Rice (preprint 53) for example, seemed too difficult to program to warrant the effort. Trees and rules thus must be written in a linear manner, using parentheses for structure identification. (Though the programmer may indent items when punching.) Any other limitations were primarily caused by my inexperience. Note especially that there is no limitation on the number of characters in a string.

CONVERSION

Until I found out about Friedman's program (preprint 14) I had considered rewriting the transformations program in SNOBOL4 -- a string processing language similar to, but incompatable with, its predecessor. It seems, however, that only the algorithm (preserved in the commentary) could be transferred as SNOBOL4's increased capabilities allowed a much more efficient approach to tree parsing.

OPERATING SYSTEMS

While it would be very nice to be able to generate sentences in a time-sharing environment, I feel that the languages currently available and especially the amount of work that must go into interfacing programs with operating systems preclude any such effort at the present time. One solution is to have full control over a small computer, however, this may excessively limit the size of the program. While there doesn't seem to be any clear-cut answer, I seem to be reluctantly choosing the power and nit-picking of the large operating system so as not to limit the programs. I hope somebody convinces me that I am wrong.

Copenhagen, 28th August 1969.

It may be too late for you to include this in your summary, but now it's written I send it anyhow.

The first step in the investigation reported in my paper, CA 3.3, is a program for sorting a text into words and word delimiters, and thus qualifies as a data processing program. In my view, this program presents the features you are interested in to a higher degree than the following programs which are datamatically much simpler, involving only manipulations of the numerical codes for words and other symbols determined by the first program. - Linguistically of course the later programs contain all the essentials.

The datamat available to me at Copenhagen University is a GIER (Danish make) with a central store of 4096 40-bit cells and peripheral store of ab. 300 000 cells. Programming is done in Algol with extensions which make the single bits of each word easily accessible. The datamat has no operator, but is available to the personnel of several institutes - which undoubtedly contributes to more frequent technical breakdowns than comparable operator-managed datamats have.

The text was obtained on 6-position paper tape without parity check, and it took some ingenuity to convert it to the usual 8-position tape. All the same it is much cheaper to get the text on these printing machine produced tapes than to code them anew.

The conversion was made not to flexowriter code, but with letters coded in alphabetic order (a=1, b=2 etc.) and other symbols with values from 32 upwards. A word is defined as a sequence of letters at most interrupted by a lower case symbol after the first, and each word found in the text is first stored in an array (of length 60 to be on the safe side). When a non-letter symbol is found the word is then converted to storage form: 5 letters are placed in the first 25 bits of a cell; the next two bits indicate whether the word has no more than 5 letters, and if not whether this is the first part of the word or a latter; 12 bits are left empty if it is the first part of the word, else two more letters are stored, and one bit indicates whether it is the end of the word or not.

Now comes the dictionary look-up. With a word stored like described the alphabetic ordering coincides with a numerical ordering when cells are interpreted as integers (the bit which in integer mode indicates sign is always left empty, i.e. as +). The dictionary is stored in an array of length 3800 to allow room for other variables including the word array of length 60 mentioned above; it is numbered 201 - 4000 for reasons explained in the paper. Each new word found is stored under the first vacant number, and every occurrence of it is indicated by this number in the output.

The alphabetic ordering is taken care of by list processing: the vacant 12 bits in the first part of a word is used to store the number of the next word in alphabetic order. To avoid having to go through the whole dictionary an index is kept of initials indicating the number of the first word with each initial. (Some reduction of search time could undoubtedly be obtained if the initial s were subdivided by the value of the next letter.)

The output of the program consists of the dictionary, number and letter sequence for each word, ordered either by numbers or alphabetically, and the processed text string, words given by their number above 200, other symbols by their number below 100, depending on the value of the last case symbol. (A space which only separates two words is suppressed; other possibilities of reduction do not present nearly the same reduction of space requirement.)

In this way, the central store will hold a dictionary of ab. 2500-3000 words (words up to 5 letters take one cell, with 6-12 letters they take 2 cells, 13-19 3 cells etc.) which in a unitary text will hold all but the very infrequent words.

The program builds heavily on the type of datamat used, only the most general principles will be transferable. Some slight alterations have benn necessary to enable the program to be run on another datamat of the same make which is operator-run (but still totally without time sharing or similar devices). I cannot to any degree of accuracy assess the initial time used for programming, which was not excessive, or that for debugging, which was considerable. Both parts of the work were done over a long period in between other work.

Gustav Leunbach

Gustav Leunbach

Answers to questionnaire

Type of Project:	Modelling of Linguistic System
Language:	Assembly language, IBM 360/65 I, 500K (h.s.)
	The computer used works in a time sharing set up, in which we are one of a number of users;
	No conversational methods are used.
Choice of Language:	Owing to the fact that 50% of the central core storage is permanently occupied by the time sharing system, the space available to our program is only 200 to 250 K; this is barely sufficient; hence, in order to compress the job as much as possible, the entire application was programmed in assembly language. The formulation of the problem and the program, therefore are in many ways influenced or, rather, determined by the characteristics of the particular machine.
Development time:	The flow charting, defining of algorithms, and linguistic research had all been done previously, in three years work, for another machine (GE 425); the time required to remodel the entire application for use on the IBM 360 was approximately 3 months. - The linguistic approach and the algorithmic formulations it requires and makes possible are highly unorthodox and, therefore, not at all suitable for formulation in an existing high-level language. The system works without vocabulary look-up; the sentences to be analysed are input on punched cards; such storage of words as occurs during the analysis procedure, is achieved by numeric code.
	There are no character manipulation subroutines; input and output definition is in IOCS. No external storage is used.
	The program is thoroughly defined by the sequence of operations determined by the linguistic procedure.
	Since the application is exclusively experimental, there is continual exchange and modification of both program and algorithms; and the program was written, from the outset, with this in mind, i.e. allowing for easy alteration in many areas.
	Notes comprehensible only to the programmer who devised the program.

Size of Program: 3200 instructions, no commentary.

The model 360 we are using disposes of approximately 180 machine instructions; the Multistore program employs no more than 30 of these (of which about 6 or 8 could be reduced to others, so that the total of used instructions could be brought down to approximately 22, or 12% of the instructions available in the machine).
This is a typical symptom of the situation of linguistic, artificial intelligence, artificial perception, etc., programming in general: the machines actually available are far too complicated, i.e. they can do innumerable things which are not needed in that kind of program; on the other hand, machines specially designed for these tasks would have to have larger central cores.
No doubt processing times could be greatly shortened on special purpose machines.

Time sharing: Yes. If we had a console in our office, it certainly would save time.

Job Control: Since the computer has to be used by other people and for other tasks as well, one has to accept job control; if we had a computer exclusively for this particular use of ours, we should do away with job control.

Change of Machine: Since the program was to some extent determined by the particular machine (capacity, byte configuration, etc.) we are using, it is not transferable to another type. Being purely experimental, this was <u>not</u> an objective.

Language: Yes. The requirements being so very specific (see above) programming in a machine oriented language is essential.

Teaching: No.

Linguist Programmers: No. The analytical work to be done to understand the workings of natural language is still so enormous that they should not scatter their attention and efforts; they should, however, have fairly clear ideas about what can and what cannot be implemented on a computer and, above all, how minutely all formulations of linguistic rules have to be defined, if they are to work satisfactorily on a computer.

Lloyd Rice
1929 12th Street - Apt. A
Santa Monica, California 90404
August 26, 1969

(preprint number 53)

In reply to some of your ideas expressed in "Metaprint", I am sending a brief history of the phonological testing program PHONOR (see preprint #53). The program has been through several translations and parts of it have actually run on two machines while other parts have not yet been coded.

The project began about a year ago, when a fairly simple program was written in Super Basic on the Tymshare, Inc. timesharing system. That program accepted a single test form, placing it in a binary matrix of feature values. Rules were written directly in Super Basic coding, performing the desired operations on the bit matrix. Later in the school year we decided to try to set up a similar system on the IBM 360/91 on campus and the Super Basic program was rewritten in PL/I. This program was still simply a rule executor and the rules had to be coded in PL/I. Difficulties with the IBM system led to the abandonment of this project. There were two main causes here which lead into the current system called PHONOR. First, I was completely turned off by the IBM system performance (91 means 91% down time). The more important reason, however, is that I wanted more flexibility in the scope display than the primitive batch job system allowed. During the time the program was being rewritten in PL/I, I was thinking more and more about a better system of rule specification and input than coding in a standard computer language, not very suitable for a linguistic researcher to use the system. Some early thinking about the string matching process and a gradually improving knowledge of Chomsky and Halle's <u>SPE</u> led to a rule compiler algorithm which accepted a string of text stating the rule using a notation quite similar to the <u>SPE</u> format and produced as output a list of matching process operations. I soon realized that these matching operations could be coded and stored fairly compactly as they were produced by the compiler and then read by a separate rule interpreter system which contained the test matrix and performed the matching operations in the order in which the compiler had stored them. This led to the present system written for the LINC-8 in our lab.

Input to the compiler will be either from the teletype or from a specified file on disk or mag tape. The input rules may be displayed on the scope in a two-dimensional format very close to the SPE formalism. This input may be edited, compiled or saved in a file. When the interpreter is loaded (by a single command to the compiler system) the most recently compiled set of rules is loaded. Operation of the interpreter is under complete interactive control of the linguistic researcher at the console, who may enter test forms, specify which rules to apply and set or reset flags for various printout options as the interpreter runs. The compiler may also be recalled at any time.

I have not added substantially to the basic compiler algorithm since writing the conference paper. I have worked out a subroutine generation system to take care of the case mentioned in the last paragraph. Actually most of the coding in the compiler is (will be) concerned with more mundane housekeeping tasks such as input text manipulation and setting up storage for the coded output.

As the program nears completion, I will definitely have clearer documentation of its structure and capabilities. I tend to avoid this as most programmers do unless I can get it done while I'm in the mood of blowing my horn (as now). Then it flows out pretty well.

I hope to remain responsive to suggestions as the program is used and desire to make it available as widely as possible.

Sincerely yours,

D. Lloyd Rice

D. Lloyd Rice

DLR:ksc

The heart of this system is a rule expression language consisting of operations to be performed on the string of phonological units stored in the test matrix. These operations are described in the paper using the PL/I language and comprise push-down stack operations, unit match instructions, matrix modification instructions and various forms of branch instructions. The system actually consists of two parts; I) A compiler, which reads the rules as they are entered and translates them to the rule expression language, and II) An interpreter, which contains the test matrix, accepts a test string from the console and interprets the rule expression language, modifying the test matrix as indicated by the rule coding.

PHONOR is now being written for the Digital Equipment Corp. LINC-8 with two linctape units and 8K of core memory. The interpreter is written in PDP-8 machine language and is now completed. The compiler is being written in LINC LAP-6 assembly language and will be running sometime in October, 1969. One memory field (4K) is dedicated to storage of the rule expression coding when the interpreter is running. I expect to get 30 to 40 average sized rules in the memory field. Additional fields of rules may be stored on Linctape and read in under program control. The present system has an upper limit of 128 rules.

One item described in the paper which the present system will not support is the notation "X", meaning any string of units not containing the boundary symbol "#". This would require a more complex matching algorithm than I have yet worked out. If it appears that such a notational device is useful it will be considered as a future extension. I hope to be able to include in the near future the capability of handling indexed disjunctions (angle brackets in Chomsky and Halle, SPE). This brings up a number of questions relating to disjunctively ordered rules (as in SPE) and the exact sequence of matching units within a rule. PHONOR treats disjunctions somewhat differently than the system of SPE in that computational efficiency is given priority over descriptive efficiency. I think it is a shortcoming of the current ideas on descriptive simplicity in a grammar that dynamic computational simplicity is not taken into account. It is my hope that future use of the PHONOR system will help in setting up new models for overall operational simplicity in the phonological component.

For more information on this system, write to

D. Lloyd Rice
Phonetics Lab, Humanities Bldg. 1110
UCLA
Los Angeles, Ca. 90024

DEPARTMENT OF COMPUTER SCIENCE
CORNELL UNIVERSITY
ITHACA, N.Y. 14850
Telephone (607) [redacted]
25--4117

PSON HALL

August 18, 1969

(preprint number 4)

I refer to your metaprint entitled "Computerized Linguistics". For your information I should like to answer the questions which you raise in so far as they apply to the SMART document retrieval system:

1. The SMART system is information retrieval oriented.

2. The system is programmed for a batch processing computer (IBM 360 model 65) largely in Fortran IV, with some of inner routines and executive programs in assembly language.

3. The choice of language was determined by the programming systems available with our computer and the preferences of the programmers.

4. The planning, flowcharting, and programming took approximately three years from 1961 to 1964, and a total of approximately 10 man years.

5. The total number of programming steps (assembly language instructions) is approximately 150,000.

6. The program is not easily transferrable onto another machine.

7. For many years I have been teaching a graduate course entitled "Automatic Information Organization and Retrieval" in which linguistic analysis procedures are used.

I should be glad to participate in the panel session if it is held within the first couple of days of the Conference (since I must leave early). I shall be glad to amplify on the comments given above.

Sincerely,

Gerard Salton
Professor
Computer Science

GS:rp

I - PROJECT Mechanical translation

The program is used both for actual processing and for testing linguistic models.

A complete program is running on an IBM 7044 computer (32K memory) and a new version is being written for the IBM 360-67.

II - LANGUAGE

Programming for the 7044 were written in MAP (macro assembly language). The program consists of eight steps, along with a supervisor embedded in the IBSYS (IBJOB) system which interfaces the different programs with each other and with input-output devices. This is, of course, a batch-processing system.

In the new program, the most important algorithms, which have to be very efficient, will be written in assembler language. Auxiliary programs will be written in PL1. This program must run both under conversational mode (using Cp-Cms system) and batch-processing mode. Conversational mode will be used for debugging and for testing linguistic models, while batch-processing will only be used for production.

The language choice never influences the problem defination.

III - STRUCTURE of the program

The program is composed of eight different steps, each roughly corresponding to a particular linguistic model. These are:

1 - pre-editing
2 - dictionary look-up
3 - morphological analysis
4 - syntactical analysis
5 - tree transformations (intermediate language)
6 - syntactical generation
7 - morphological generation
8 - post-editing

Each step requires:

1 - the program itself
2 - the input text (which is the output of the previous step)
3 - linguistic parameters: grammar and lexicography
4 - the output text.

The last three are encoded to preserve program efficiency. Grammars, for example, may be pre-compiled by a special subroutine.

It is also necessary to provide auxiliary programs; giving input, output, and if necessary, intermediary results a human-readable form.

Thus, we need to write two different types of programs, the processor -- which must be very efficient and is usually quite short -- is written in assembler language. The auxiliary programs -- which need not be particularily efficient, but must be easily modifiable -- are written in a problem-oriented language, PL1. The latter represent 60% of the programming work (including compilers, text file updating, dictionaries, etc.)

IV - TIME REQUIRED

This depends on the nature of the step. In the case of syntactical analysis, probably the most important, the following roughly holds:

statement of proglem: about two or three years
defining data structures and system programming: six months
programming and debugging the algorithm: one year
programming and debugging auxiliary programs: one year
computer time for program debugging: ten hours (7044)

The complete 7044 program, including all eight steps, contains about 65 000 machine instructions, 20 000 for the program, 45 000 for auxiliary routines.

V - PROGRAM CONVERSION

After the 7044 programs were debugged, we began changing to the 360-67. We are trying to convert all algorithms directly. The most important changes are relative to data managment. We had many problems with tape devices for the files and feel that the direct-access capabilities of the newer machine will prove very useful. In writing the first program, we were very cautious about program efficiency. While this is, of course, important, it did become very time consuming for the linguistic debugging (of grammars) and dictionary updating. This was partly due to batch-processing. With the new computer, we shall always use conversational node for debugging. The program thus must be executable in both conversational and batch modes. The most important problem is to make the files compatable under both systems.

PL1 seems to give us all the power we need, but we intend to limit iss use to auxiliary programs.

I think it is important to speak a little about artificial languages for linguistics. We were obliged to define special languages for this purpose.

In some cases, we wrote a compiler; while in others, such as tree transformation, we used a sophisticated macro processor. Macro assembly is very attractive -- the operations being easy to define, describe, and modify. In our case, language defining and macro writing took only three months. Unfortunately, macro assembly is very slow and, in the case of the 360, not sufficiently powerful. We were thus obliged to write our own compiler, instead of using the IBM software directly.