PURDUE UNIVERSITY

COMPUTER RESEARCH PROGRAM

PURDUE COMPILER

GENERAL DESCRIPTION

Sylvia Orgel

Copyright July, 1958

PURDUE RESEARCH FOUNDATION

SUMMARY

The Purdue Compiler, a mathematical language compiler, is herein defined and its structure outlined. The compiler generates a machine language program from a flow chart written in the form of statements for the Datatron Computer. This can result in a decrease in the coding staff of an installation.

The "matrix logic" of the compiler is of prime importance. A symbol scanner examines pairs of symbols, determines their admissibility from an entry in a matrix, and triggers a unique generator which compiles the symbols' meaning. This allows for a simple scanner structure regardless of the complexity of the string of symbols being scanned. The matrix which is easily augmented or changed, is the core of the compiler.

The compiler uses:

1. The Main Computer
2. Two Magnetic Tape Units
3. Paper Tape Input
4. Paper Tape Output
5. Flexowriter Output
6. Card Input (Optional)

CONTENTS

	Page
PREFACE	1
INTRODUCTION	2
THE COMPILER LANGUAGE	5
A. Elements	5
1. Constants	5
2. Variables	5
3. Operators	6
4. Punctuation	7
B. Functions	7
C. Operands and Expressions	8
D. Statements	9
1. Substitution Statement	9
2. Linkage Statement	10
3. Relation Statement	11
4. Halt Statement	12
5. Output Statement	12
6. Input Statement	13
7. Extension Statement	13
THE PROCESS OF COMPILATION	14
THE STRUCTURE OF THE COMPILER	17
A. Introduction	17
B. Section I	20
C. Section II	22
D. Section III	25
EXAMPLE	28

APPENDICES

A. Two Digit Compiler Code 30

B. Admissible Symbol Pairs 31

PREFACE

In September, 1955, four members of the Purdue University Computing Laboratory [Mark Koschman, Sylvia Orgel, Alan Perlis and Joe Smith] began a series of conferences to discuss methods of automatic coding. [Joanne Chipps joined the group in March, 1956.] A compiler, programmed to be used on the Datatron, was the goal and result.

These conferences continued until June, 1956, at which time an outline of the basic logic, language and general aspects of the compiler was completed. Further work on the logic, programming and coding of the compiler was continued by J. Chipps and S. Orgel. The compiler was completed in the summer of 1957.

This report is the first part of a complete description of the Purdue Compiler. Other parts will include a modification and revision of the current handbook, the detailed information of the structure of the Compiler, and the Compiler flow diagrams.

INTRODUCTION

A compiler may be defined as a program which satisfies the following conditions:

1. It provides direct machine translation of flow charts into a program.
2. It has the ability to translate into machine language any program which could have been coded into machine language. (This is trivially satisfied if the compiler can accept machine language.)
3. It automatically allocates machine storage.

The motivations leading to the consideration of a compiler are obvious to any one who has ever written at least one program. Nevertheless, they are summarized briefly as follows:

1. Coding is a highly repetitive technique and hence capable of automation.
2. People who propose problems should program them, but should not be required to code them.
3. The ratio of the time required to flow chart a problem to that for coding it should be much larger than one. Currently using manual coding, it is much smaller than one.

The usual sequence of steps in solving a problem with a digital computer is as follows:

1. Definition of the problem. A problem which is not well defined is a problem which cannot be solved. In order for a computer to be of use in solving a problem, the problem must be well defined and thoroughly understood by the person making use of the computer.

2. Logical solution of the problem. This is generally done by means of a flow chart, that is, a diagrammatic device for representing those logical operations necessary to arrive at a solution of the problem. This logical solution is common to all methods of problem solving.

3. Coding or translation of the program into machine language. Using the flow chart as a guide, the logical operations are written in the language of the computer (machine language). The sum total of all the coding for the solution of the problem is called a routine, or a program.

Definition and logical solution of the problem should be the domain of the person who proposes the problem with the aid (if needed) of a programmer or analyst. Coding requires a person conversant with the language of the computer used to obtain the solution of the problem.

An automatic coding system (or compiler) uses the computer to perform the detailed coding of the problem. Thus, as the flow chart is the guide to the coder, it should also be the guide (or input) to the compiler.

A flow chart is a graphical representation of the analysis of the problem into a logical sequence of operations. The operations fall into the following general categories:

1. Communication of information - input to and output of the problem.

2. Arithmetic Computations of the problem.

3. Decisions within the problem.

4. Linkages among the above three.

The elements of a flow chart are constants, variables, operators, punctuations, functions, and some means of

representing the different operations (statements).

The Purdue Compiler was developed primarily for scientific and engineering applications; therefore, the structure of its input was chosen to be similar to mathematical formulation. The Datatron can accept, as input, only decimal digits. Thus, in order to extend the language, a two digit code was assigned to each letter, digit, punctuation, and operator. A total of 48 characters make up the language - 26 letters, 10 digits, +, -, ,, *, (, .,), =, /, ≤, <, ", and space. The two digit code is given in Appendix A.

The compiler is written for the Datatron but can be used only if the following auxiliary equipment is available:

1. Two magnetic tape units.
2. Paper Tape input.
3. Flexowriter or high speed punch output.

Card input can also be used but is not mandatory.

There are two versions of the compiler, one for machine (automatic) floating point and the other for programmed floating point.

THE COMPILER LANGUAGE

A. Elements

1. Constants

Constants are represented in two forms - fixed point integers and floating point constants. Fixed point integers can contain at most ten digits and are positive. Floating point constants are positive, must lie in the range of 10^{-50} to 10^{49} and consist of, at most, eight digits.

	General Form	Examples
Fixed Point	A sequence of 1 to 10 decimal digits	324 7 123456789
Floating Point	A sequence of decimal digits including a decimal point, a b (base 10) or both.	234. = $.234 \times 10^{+3}$ 7.92b7 = $.792 \times 10^{+8}$ 12b(-3) = $.12 \times 10^{-1}$

2. Variables

Two kinds of variables are required since the computer arithmetic functions in both the fixed and floating point modes. Each variable is designated by a letter and a subscript. The subscripts, written on the same line to the right of the variable letter designation, must be fixed point variables, fixed constant or composites of both delimited by parenthesis. Fixed point variables (indices) are represented by the letter i. Floating point variables have two representations, y and c. This aids in external differentation ~~since~~ but internally they are treated identically.

	General Form	Examples
Fixed Point	in[1]	i3
	ii...n	iii2
	i(...)	i(3 x i1)
Floating Point	yn	y27
	yi...in	yi3
	y(...)	y(i1 + i2/2)
	cn	c335
	ci...in	cii12
	c(...)	c((i2 x 3) + 22)

3. Operators

Operators are classified as arithmetic, both unary and binary; substitutional; and relational.

		Symbol	Meaning
Arithmetic			
	Unary	a	absolute value
		(- ...)	negative of
	Binary	+	addition
		-	subtraction
		x	multiplication
		/	division
		*	exponentiation
		p	integral power
		b	base 10 - defined only for fixed pt. constants
Substitution		=	replacement
Relational		≤	less than or equal to
		$<$	less than
		z	equal to

---

(1) The letters n, m, and k will be used to denote integers.

Henceforth, ω will be used to designate any binary arithmetic operator and λ, any relational operator.

4. Punctuations

Symbol	Usage
(	} beginning & end of operand
)	
.	with floating point constants
,	in relation statements and to separate operands in extensions
"	beginning and end of extension
f	end of statement

B. Functions

Functions are represented by the symbol pair ne – the n th extension (subroutine) to the compiler. They are delimited by quotes and the parameters are separated by commas.

General Form	Example
"ne, _, _,..., _"	"200e,y1" = $\sqrt{y1}$
	"230e,c2" = $\log_{10} c2$

The arithmetic mode of the result of the function is determined by the first digit of the three digit extension number. If even, the result is in floating point; if odd, in fixed point.

C. Operands and Expressions

1. Operands

   a. A variable or constant is an operand.

   b. If v1 and v2 are operands, (v1 ω v2) is an operand. (ω is a binary arithmetic operator.)

   c. Extensions are operands.

2. Expressions

   a. If v1 and v2 are operands, v1 ω v2 is an expression.

   b. All operands are expressions.

3. Arithmetic

   a. If an operand is a variable or constant, its arithmetic is that of the variable or constant.

   b. If one variable in an operand, not including extensions, is in floating point form, the operand is floating point.

   c. The arithmetic of extensions is discussed on page 15.

Examples	Meaning
i2 x i1	Fixed point multiplication
y1 p i1	$y1^{i1}$
y1 p c2	Not permitted since c1 is not a fixed point integer
(c3 * y4)	$c3^{y4}$
y23 * i7	$y23^{i7}$ (y23pi7 would be preferable)
(y2 x i1)	Convert i1 to a floating point number and multiply by y2

D. Statements

The input to the Purdue Compiler is a statement program. The statements are the equivalent of a flow chart written in the language of the compiler. The general format of all statements is:

a. The first symbol of the statement is an integer, the statement number. This will be designated by k.

b. A letter following the statement number, is the statement designation.

c. The last symbol of the statement is a finish pulse f.

The statement number uniquely identifies each statement relative to a problem. Since statements are compiled as entered into the computer, the statement numbers need bear no relation to the physical ordering of the statements.

The statements types that are, at present, incorporated into the compiler are discussed below.

1. Substitution Statement.

Format	Example
k s vn = an expression f	3 s y2 = cl + (yl x y7) f

A substitution statement is identified by the letter s. The value of the variable to the left of the equal sign — designated here by vn — is set equal to the result of the expression. That is, the contents of the cell designated by the left hand variable is replaced by the result of the right hand expression. The result will be stored in fixed or floating point according as the variable on the left hand side is a fixed or floating point variable.

If the variable on the left is fixed point and the expression of the right is floating point, the result will first be computed in floating point and the greatest integer in the result will be

the value of the fixed point variable. Thus, if the result is ± 2.893, the fixed point number stored will be ± 2.

Examples of Substitution Statements.

Statement	Meaning
1 s yl = y2 f	yl is repalced by the value of y2.
2 s il = yl f	The greatest integer in yl replaces il.
7 s y4 = i3 f	Convert i3 to floating point and store in y4.
3 s il = il + 1 f	Add 1 to il and store in il. This example illustrates the point that a substitution statement is not an equation but a command to replace a value.
24 s il = 3 x yl f	Convert 3 to floating point, multiply by yl, obtain the greatest integer of the result, and replace il by this value.

2. Linkage Statement.

Format	Example
k g n f	22 g 3 f
k g i...in f	17 g i4 f

Linkage statements are used to modify the natural order of statements and are identified by the letter g (i.e., g for go). They link the preceeding statement to the statement whose number is n or i...in. The use of fixed point variables as statement numbers in linkage statements allows for variable connections between statements in a flow chart.

3. Relation Statement.

Format	Example
k r g n, r operand λ expression f	2rg7, r yl z cil f
krgi ...in, r operand λ expression f	34rg i2, r il < l f

Relation statements provide conditional linkages — the condition being the satisfying of the relation between the operand on the left and the expression on the right of the relation operator λ. If the condition is satisfied, the next statement to be executed is that one whose number follows the g. If the condition is not satisfied, the statement of the program immediately following is executed.

Note: The left side of the relation must be an operand.

If the arithmetic mode of the two sides of the relation are not the same, the arithmetic of the relation operator is determined by the right side.

Examples of Relation Statements.

Statements	Examples
7rg 12, r (yl + y2) z c3 f	Go to statement 12 if the value of yl + y2 is equal to c3; if not go to the ~~next~~ statement which follows.
7rg 12 r yl + y2 z c3 f	Not admissible since yl + y2 is an expression but not an operand. (must use parentheses)
2rg i2, r il < yl f	If il converted to floating point is less than the values of yl, go to statement i2.
225rg 107, r y2 < il f	If the greatest integer in y2 is less than il, proceed to statement 107.

4. Halt Statement

Format	Example
k h f	251 h f

Halt statements cause the computer to stop. Pressing the continuous button on the console causes the program to resume ~~at~~ with the next statement. Therefore, a halt statement can be used as a terminal or temporary stop.

5. Output Statement

Format	Example
k o v n f	17 o y21 f
k o v i ... in f	314 o cil f
k o v (...) f	7 o i(i2 + 7) f

Output statements, identified by the letter o, result in the print out of two words (ten digits and sign if negative). The first word whose format is kkkCOdCmmm identifies the statement number, k, of the output statement, the variable, d = 1, 2, or 3 is v = i, y, or c, and the value of the subscript m. m = n or m = the current value of i...in or (...). The second word is the current value of the variable.

Examples of Output Statements.

Statement	Print Out
17 o y21 f	0170020021 value of y21
314 o cil f	Assuming that the current value of il is equal to 26, 3140030026 value of c26
7 o i(i2 + 7) f	If i2 + 7 is currently equal to 14, 0070010014 value of i14

6. Input Statement.

Format	Example
input in yn cn sn f	input i12 y83 c7 s41 f

Input statements are used as lead statements to each problem and have no statement number associated with them. The numbers following i, y, and c denote the maximum value of the subscript associated with each in the problem. The number following s denotes the maximum statement number used in the problem. The compiler uses this statement to allocate storage for the variable and for a statement dictionary. In the example given above, 13 memory cells are allocated for the i's (io through i12), 84 for the y's, 8 for the c's and 42 for the statement dictionary of this problem.

Input statements are used by the compiler and do not contribute directly to the compiled program.

7. Extension Statement.

Format	Example
k e "ne, __, ..., __" f	27 e "812e, 6, y1" f

Extension statements contain extensions that perform sequences of operations not leading to the definition of a single variable. Extension statements can be used for the following, provided the extensions are available.

a. To obtain format control of output.

b. To arrange p variables in numerical sequence

c. For card output.

etc.

THE PROCESS OF COMPILATION

Statements are compiled in their order of entry into the computer regardless of their statement number. The compilation process for any statement is independent of the nature and presence of other statements. It is unidirectional from right to left.

The binary arithmetic operator ω (where ω = +, -, x, /, *, or p) is found in an expression in the form $\alpha \omega \beta$/. α is the operand whose rightmost character is adjacent to ω on the left. β is the expression that starts to the right of ω and is terminated on the right by an unpaired right parenthesis, a comma, a quote, or the end of the statement.

The relation operator λ (where λ = z, $<$, or $\leq$) follows the same rules as the binary arithmetic operators. Therefore, an expression is not allowed as the left hand side of a relation statement.

The unary operator a (absolute value) only refers to the operand immediately to its right. The unary operator (- (negative of) operates either on an operand or on an expression which is terminated on the right by an unpaired right parenthesis.

Within an expression, binary arithmetic operators are compiled from right to left in the order in which the leftmost character of their left operands are found. Unary operators are compiled as they occur.

This method of compiling an arithmetic expression does not conform to the standard hierarchy of operations. The expression y1/y2 + c1 is compiled as y1/(y2 + c1) unless parenthesis are used to give it the meaning (y1/y2) + c1.

Example of Compilation Order:

yl + (y2 x (2 + i4)/ il + 1) - a (y2 p il)

Order of Compilation	Left	Operator	Right	Arithmetic Mode
1	y2	p	il	floating
2	—	a	(y2pil)	floating
3	il	+	1	fixed
4	2	+	i4	fixed
5	(2+i4)	/	il+1	fixed
6	y2	x	(2+i4)/il+1	floating
7	(y2x(2+i4)/il+1)	-	a(y2pil)	floating
8	yl	+	(y2x(2+il)/il+1)-a(y2pil)	floating

The arithmetic of extensions is determined by the extension number, $n_1 n_2 n_3$. If n_1 is odd, the result of the extension is in fixed point form; if n_1 is even, in floating point form.

The basic logic of an automatic coding system depends upon the method of translating a problem written in the language of the system into a machine language code. The structure of the Purdue Compiler was enormously simplified by attaching meaning only to an ordered pair of symbols. The scanning of the statement from left to right is accomplished by stepwise examining symbol pairs - $s_{i-1}s_i$. This method of scanning permits the psuedo-machine code for a statement to be generated in a single pass through the statement.

7

Within the compiler, there exists a matrix with one entry for each possible symbol pair. The numerical value of this matrix entry together with the symbol pair determines:

1. The admissibility of the symbol pair. A zero matrix entry indicates an inadmissible symbol pair.

2. A generator which prepares the psuedo-machine code of the symbol pair.

Associated with the matrix and generators are a multiple set of accumulators, operator registers, and current arithmetic register. The latter two are used only during compilation. There are 18 of each available. This limits the nesting of parentheses and quotes in one expression to 18.

There are thirty nine (39) distinct symbols in the compiler language. The digits 0 through 9 are herein considered as a single symbol. Spaces are ignored in scanning a statement by symbol pairs. Therefore, a space is not considered as a symbol when referring to the matrix. Of the one thousand five hundred and twenty one (1521 = 39 x 39) possible symbol pairs, only one hundred and ninety eight (198) are admissible. There are only seventy one (71) distinct generators since many symbol pairs result in similar generated codes. Appendix B lists the admissible symbol pairs, grouped according to generators.

THE STRUCTURE OF THE COMPILER

A. Introduction

The process of compilation logically divides into three sections which are translation, assembly, and display. Section I, the "translator", scans the statement program which represents the flow chart of the problem. In addition to allocating storage for the variables, it prepares a program in a psuedo-machine code for using relative and floating addresses. Section II, the "assembler", allocates storage for the program, translates it into Datatron code and optimizes the coded program by a looping routine. Section III, the "displayer", supplies to the programmer all the information needed for running the compiled program. This includes a detailed print out of the original statements and of the program, and a paper tape of the program.

The main compiler routine is composed of these three sections and an extension dictionary. In addition, there is a preliminary part which stores the statement program on magnetic tape and initiates compilation. At present there are three routines that will accomplish this aspect. They are:

1. Pretape - for use when the statement program is on paper tape in the compiler code.
2. Card Input Translation Routine - for use when the statement program is on cards. The card code, a two digit code, must first be translated into the compiler two digit code before it is stored on magnetic tape.
3. Compiler Statement Check Routine (CSCR) - for use both with paper tape and cards. This routine incorporates routines 1 and 2 , and also checks the

statements for format errors and inadmissible symbol pairs prior to compilation.

The compiler is stored on magnetic tape from four paper tapes. These are:

1. Section I
2. Section II
3. Section III
4. Extension Dictionary

The auxiliary equipment listed below is necessary if the Purdue compiler is to be used. In addition, the function of each is given.

1. Tape Unit I
   a. For storage of the original statement program in the compiler two digit code.
   b. For temporary storage of the semi-looped program.
2. Tape Unit 2
   a. For storage of the compiler routine.
   b. For temporary storage of
      i. the psuedo-machine code program,
      ii. the looped program.
3. Paper Tape Input
   a. To read the compiler into main memory for positioning on magnetic tape.
   b. For input of the statement program.
4. Card Input (optional)

   For input of the statement program.

5. Flexowriter

For detailed print out of the original statement program and the compiled machine language program.

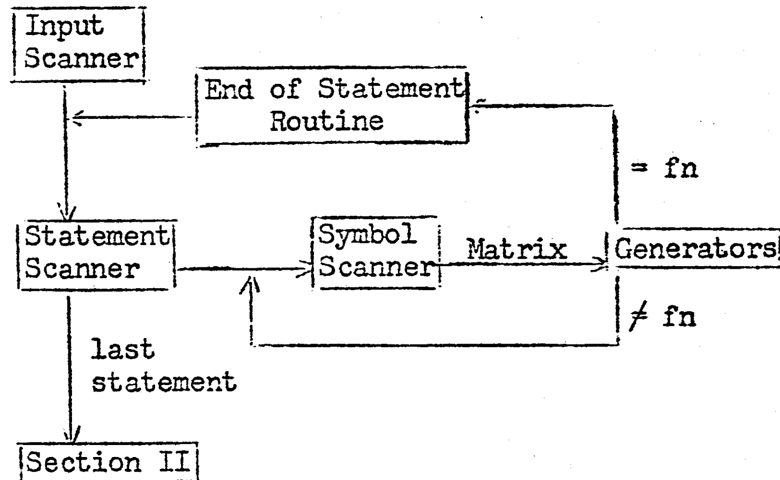
6. High Speed Punch

For punch out of the compiled machine language program.

Not the same (identically)

B. Section I.

Section I has two functions: assigning storage for the variables and the statement dictionary, and generating a psuedo-machine language program for each statement. The overall flow diagram of Section I is given below and the function of each part is discussed.



1. Input Scanner.

The input scanner examines only the first statement of the program - the input statement. Starting with the word input, the scanner proceeds from left to right assigning storage first for the i variables, then the y's, the c's, and the statement dictionary. The latter is used by linkage and relation statements. The base address of the compiled program is also determined by the input scanner.

2. Statement Scanner.

The statement scanner examines one statement at a time. It starts with the finish pulse at the end of the last statement (left f) and searches for the location of the finish pulse at the end of the current statement (right f). These two finish pulses delimit the statement. If, instead of a right f, a flag indicates the end of the statement program is detected, control is transferred to Section II.

3. Symbol Scanner.

The symbol scanner starts with the right f of the current statement and proceeds from right to left examining symbol pairs using the matrix. The matrix determines if a symbol pair is admissible and the generator associated with it.

4. Generators.

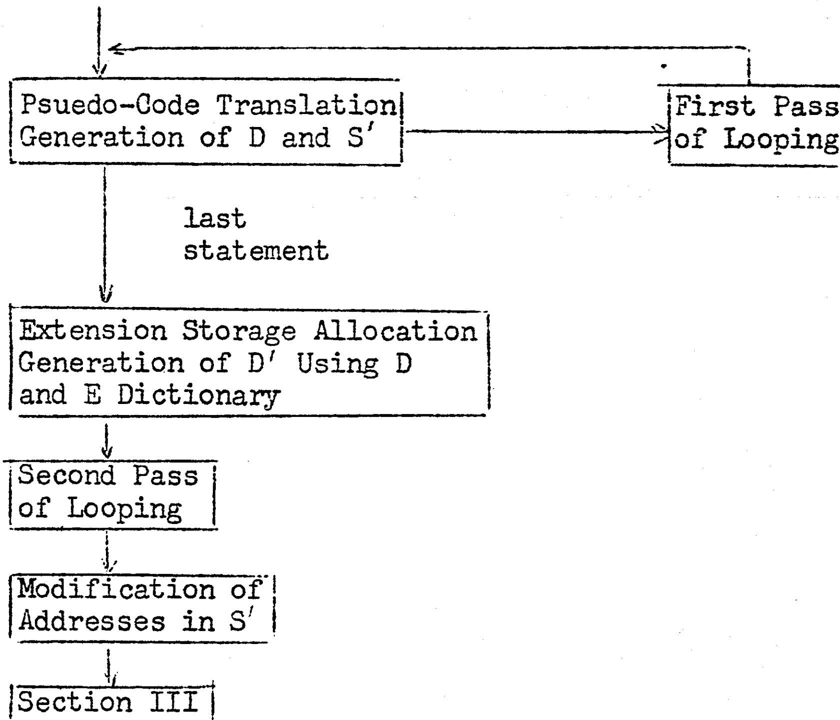
The generators compile the orders - in psuedo-machine language code - or set up the compiler parameters as dictated by the symbol pair. The exit of all generators with exception of that one associated with fn, is to the symbol scanner.

5. End of Statement Routine.

The end of statement routine inserts after the compiled psuedo-machine language program of the statement, information about the statement. This includes the statement number and information as to the location of the statement on magnetic tape in its original form. The latter is used for part of the detailed print out in Section III. The compiled psuedo-machine language program for the statement is then stored on magnetic tape and control returned to the statement scanner.

C. Section II.

The overall flow diagram of Section II is given below. Its main function is that of an assembler - starting with the psuedo-machine language code of each statement and resulting in the final looped program.



1. Psuedo-Code Translation.

In this part, the psuedo-machine language code is changed into Datatron code, statement by statement. The relative program addresses are changed to machine addresses. The first address of the program is set equal to the base address as determined by the input scanner in Section I. Two dictionaries, the internal statement dictionary, S' , and the current extension dictionary, D, are generated. The contents of D is a list of the extensions required by the program.

2. Internal Statement Dictionary - S'.

The internal statement dictionary is generated using the information inserted by 'the end of statement routine' of Section I. It is arranged according to the incoming sequence of the statements and is composed of two Datatron words per statement. The first word contains the statement number and the address of the first order in the statement program. This address is modified after the second pass of looping. This modification results from the expansion of the program by the looping routine. The second word contains the block and symbol information necessary to locate on magnetic tape the left and right finish pulses which delimit the statement.

3. First Pass of Looping.

The first pass of looping generates a semi-looped program. It divides the program and constants into blocks of twenty, with orders starting in the zero modulo twenty location and constants in the nineteen modulo twenty location. All references to constants are modified so that they refer to the 7 loop. As each block of twenty is completed, a cub order to the next block is inserted as the last order of the block. This is done, statement by statement, but more than one statement (or parts thereof) can be contained in a single block of twenty.

4. Extension Storage Allocation.

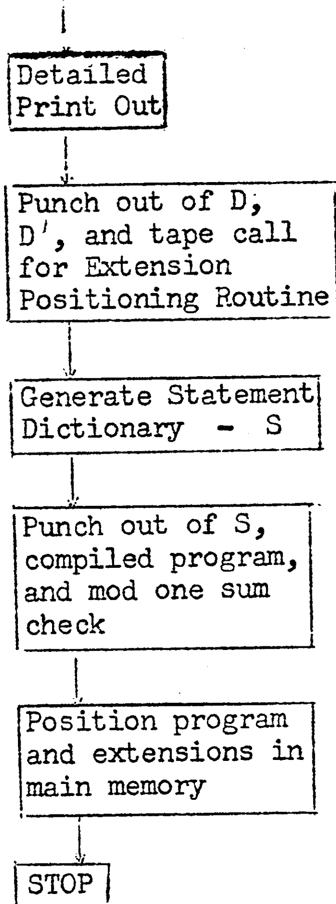
After the last statement has been semi-looped, main memory storage is allocated for the extensions. This utilizes the extension dictionary, E, on magnetic tape and the current extension dictionary, D, and generates D'. For each extension listed in D, the contents of D' is the block length, tape address, and assigned main memory address. If an extension is listed in D, but the information concerning it is not stored in E, the block length is entered by the programmer (or operator) from the computer console.

5. Second Pass of Looping.

The second pass of looping modifies the semi-looped program block by block. For change of control to extensions, the main memory address of the extension is inserted using D and D'. If an order refers to the program, its address must be modified. This is necessitated by the expansion of the program through looping. With certain orders, a simple change of address is not always possible and a block dictionary must be generated as part of the final program.

D. Section III.

The general flow diagram of Section III, the displayer, is shown below,



1. Detailed Print Out.

The detailed print out contains all pertinent information concerning the program. It is ordered as follows:

a. The addresses of i0, y0, c0, s0, and b0.

   These are the first addresses used by the variables, i, y, and c, the statement dictionary, and the program.

b. A list of the extensions in the program, their block length, main memory address, and tape address.

c. The first statement of the problem.

d. The program associated with the first statement including the addresses and their contents.

e. A repetition of c. and d. for each statement of the problem.

f. The block dictionary if one was generated in the second pass of looping.

2. Extension Positioning Routine.

This routine positions in main memory the extensions used in the problem that are stored in magnetic tape. The decisions are based on the contents of D and D'.

The extension positioning routine is employed both by the compiler and the final compiled program. The compiler uses it when positioning the program for immediate running. In order for the final compiled program to utilize it, D, D', a tape search and a tape read of the routine is punched out as part of the compiled program.

3. Statement Dictionary.

This dictionary is part of the compiled program and is used by relation and linkage statements. It is ordered

according to statement number and starts in address s0. There is one word per statement whose contents is a cub to the first order in the statement. It is generated from the internal statement dictionary, S'.

4. Punch Out.

The punch out contains the statement dictionary, the compiled program, a mod one sum check of both, and the necessary paper tape commands for reading the information back into the computer.

5. Final Positioning.

This portion of Section III is equivalent to reading in of the punched out information; i.e., the complete program. Its purpose is to permit immediate running after read-in of the data for the problem.

Example

The following example will indicate the use of the compiler with a relatively simple problem, namely, the standard deviation.

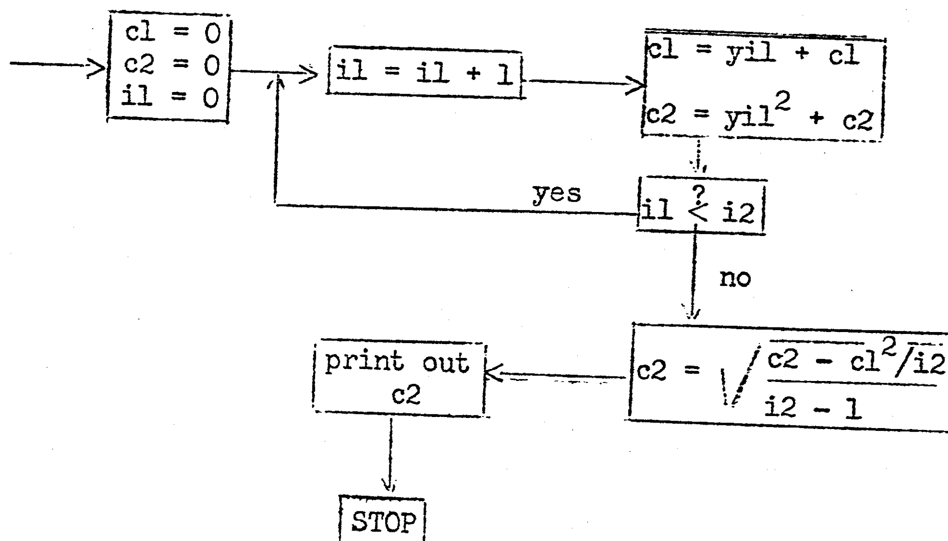
Given a set of variables x_j where $1 < j \leq n$ and $n \leq 999$, compute and print out the standard deviation. The formula for the standard deviation, s, is

$$(1) \quad s = \sqrt{\frac{\Sigma_j (x_j - \bar{x})^2}{n - 1}} \qquad \text{where } \bar{x} = \frac{1}{n} \Sigma_j x_j$$

$$\text{or} \quad (2) \quad s = \sqrt{\frac{\Sigma_j x_j^2 - \frac{1}{n} (\Sigma_j x_j)^2}{n - 1}}$$

Relabeling the variables for use with the compiler, we have il = j and yil = x_j. Since n is itself a variable, let i2 = n. cl and c2 will be used as temporary storage.

Thus, the flow diagram of formula 2 is:



The compiler statements for this flow diagram are:

```
 input i2 y999 c2 s10 f
 1 s c1 = 0 f
 2 s c2 = 0 f
 3 s i1 = 0 f
 4 s i1 = i1 + 1 f
 5 s c1 = yi1 + c1 f
 6 s c2 = c2 + yi1 p2 f
 7 r g4, r i1 < i2 f
 8 s c2 = "200e, (c2 - ((c1 p 2 )/ i2 ))/ ( i2 - 1 ) " f
 9 o c2 f
10 h f
```

APPENDIX A - TWO DIGIT CODE

Code	Character	Code	Character
00	0	25	p
01	1	26	q
02	2	27	r
03	3	28	s
04	4	29	t
05	5	30	u
06	6	31	v
07	7	32	w
08	8	33	x
09	9	34	y
		35	z
10	a		
11	b	40	+
12	c	41	-
13	d	42	,
14	e	43	*
15	f	44	)
16	g	45	.
17	h	46	(
18	i	47	=
19	j	48	/
20	k	49	≤
21	l	50	<
22	m	51	"
23	n		
24	o	99	space

Appendix B

The admissible symbol pairs are listed below grouped according to generators. n designates a digit.

n g	n +	c i	i n
n ,	n -		
n r	n x		
n h	n /	c n	i i
n)	n p		
n e	n *		
n s		c (	i (
n "			
n o	a i		
		e ,	o y
		e "	
n n	a y		
n b	a c		o c
n .		f n	
	a "		o i
n f		g n	
	a (		r n
n z		g i	r .
n ≤			
n <	b n		
		h f	r y
			r c
n =	b (		r a

r i	= a	= i	= y
	+ a	+ i	= c
	- a	- i	+ y
r "	x a	x i	+ c
	/ a	/ i	- y
	p a	p i	- c
r (	* a	* i	x y
	(a	(i	x c
			/ y
r g			/ c
	= "	= n	* y
	+ "	= .	* c
s i	- "	+ n	(y
	x "	+ .	(c
	/ "	- n	
s y	p "	- .	
s c	* "	x n	z n
	("	x .	z .
		/ n	≤ n
y n		/ .	≤ .
	= (	p n	< n
	+ (	* n	< .
y i	- (	* .	
	x (	(n	
	/ (	(.	z (
y (	p (		≤ (
	* (		< (
	((		

z y
z c
≤ y
≤ c
< y
< c

z a
≤ a
< a

z i
≤ i
< i

z "
≤ "
< "

, r

, n
, :

, i

, a
, y
; c

, "

, (

) f

) +
) -
) x
) /
) p
) *

))
) ,
) "

) z
) ≤
) <

) =

(-

.

" f

" "
")
" ;

" +
" -
" x
" /
" p
" *

" z
" ≤
" <

" n

. f

. "
:)
: ;

: n
: b

: +
: -
: x
. /
: p
: *

. z
: ≤
: <