# FAST* (model 402)
# A FORTRAN Type Compiling System for FACOM 222

MAKOTO TSUJIGADO** AND TAKESHI MARUYAMA**

## 1. *A Survey of the System*

### 1.1. *Introduction*

FACOM 222 FAST (model 402) is a FORTRAN-type compling system, prepared for the computer FACOM 222 with a core storage of 4,000 words, no drum storage, 2 magnetic tape units, one paper tape reader, one paper tape punch and one line printer. The source language is nearly the same as that of IBM 7090 FORTRAN, but is deficient in EQUIVALENCE statement and the statements which control magnetic tapes and magnetic drums.

The fundamental objectives in designing this system are as follows.

(1) To shorten the compiling time, the time of scanning a source program is to be reduced as much as possible, and the relative coded program similar to machine code is to be produced directly without the intermediate phase of symbolic assembly.

(2) More than two subprograms compiled at different times are to be combined and executed as a complete program.

(3) Following simple control informations, any sequences of compilations and executions are to be done.

(4) Arithmetic computation of complex numbers is to be available.

(5) Lukasiewicz expression (Polish prefix notation) is to be utilized in compilation process.

Average compiling time for each statement is 5 sec. Although the number of identifiers of operands in each subprogram is rather limited, some users have accomplished massive computations, using CHAINJOB and magnetic tape routines coded by hand.

### 1.2. *On the available statements and limitations*

The available statements in this compiler are the same as those of FORTRAN II, except EQUIVALENCE. In addition, a declarative statement COMPLEX is included.

Each $I_i$ of END ($I_1$, $I_2$, $I_3$, $I_4$, $I_5$), respectively, corresponds to five al-

This paper first appeared in Japanese in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 4, No. 5 (1963), pp. 241–248.

* Short for Fuji Automatic Sentence Translator.

** Program Section of Computer Division, Fujitsu Limited, Tokyo.

teration switches on the console. If $I_i=0$ or 1, $I_i$ is independent of the switch status, and if $I_i=2$, the compiler asks the status of the switch. The functions of the switches are as follows.

ASW 1=DOWN: On the completion of the compiling phase, the object program may be loaded, that is, the computer halts after loading the object program's loading routine (relocatable loader) into the core storage from the system tape. By hitting the start key, loading operation of object programs is conducted.

ASW 1=UP : After compiling a source program, the status is ready for compiling another source program.

ASW 2=DOWN: Lists are taken.

ASW 2=UP : No lists are taken.

ASW 3=DOWN: If the name of an operand in the coming-in subprogram is identical to that in the current subprogram having already been compiled, both occupies the same location.

ASW 3=UP : Operands in the coming-in subprogram occupy different locations in the current subprogram, no matter whether the names of the operands are identical to each other. It is normal that ASW 3 is UP.

ASW 4=DOWN or=UP status determines whether the object program is punched on a paper tape or not.

ASW 5 must always be UP.

At present, the limitations on the source program are rather severe.

## 2. *Compiler**

### 2.1. *General descriptions of the compiler*

(1) The first part (3,245 machine words) reads source program in the core storage, assigns a relative address to an operand forming necessary tables, encodes an operator and writes out all informations on to the magnetic tape No. 2.

(2) The second part (about 700 words) scans the information on the magnetic tape No. 2 and outputs FORMAT codes.

(3) The third part (2,775 words) produces the object program from the information on the magnetic tape No. 2.

(4) The fourth part (about 800 words) outputs constants with relative locations of statement numbers in the object program. These four parts are stored respectively in four physical blocks on the magnetic tape No. 1 (system tape), and are loaded successively in the storage.

* cf. the general flow chart Fig. 1.

2.2. *On the formula translation*

The object program in the machine language is produced from the Lukasiewicz expression.

Lukasiewicz expression is defined as follows:

(1) An operand is a Lukasiewicz expression.

(2) If $Z_1, Z_2, \cdots, Z_n$ are any Lukasiewicz expressions, and if $F_n$ is an $n$-ary operator, then

$$F_n, Z_1, Z_2 \cdots Z_n$$

is a Lukasiewicz expression.

When an arithmetic formula "variable=expression" is replaced by "expression=variable" in the core storage, the left hand side is the first operand and the right hand side is the second, and hence the arithmetic formula has no distinction from the expression in the translating process. Placing the Lukasiewicz expression in the reverse direction, alloting 1 for an operand and $1-n$ for an $n$-ary operator, we make partial sums. If the partial sums from the left end are all greater than zero and the last partial sum is 1, then the expression is valid. This is one of the checking procedures. The object program is produced by scanning the sequence of symbols left to right and searching for the corresponding operand to an operator encountered.

Subscripts are first processed, being optimized, in a statement.

In the compilation process, about 100 kinds of errors are watched, 69 in the first part, 2 in the second, 21 in the third and 7 in the fourth. Once an error is detected, processing of the statement is interrupted to display the error message, and immediately processing of the next statment is entered so that the compilation can be continued.

## 3. *Object Programs*

### 3.1. *The output format of an object program*

A (sub)program is translated into the sequence of relocable codes similar to machine instructions, headed by its name and by the names and the relative codes of other subprograms, if any, called for by this program.

This sequence is followed by

(1) the statement numbers to the relative addresses correspondence table,

(2) control informations for temporary storage allocation,

(3) sizes of data and program areas,

(4) the arithmetic statement function to its relative address correspondence,

(5) names of library functions referenced to,

(6) constants and their locations and order codes whether to proceed

to next compilation or to load and execute object programs (in the latter case, moreover, followed by informations to load necessary routines for execution).

They are punched out on the paper tape, and, if so desired, together with the names of variables to their relative address correspondences, may be listed on the line printer.

### 3.2. *Loader*

The loader loads in one main program and, if any, subprograms compiled independently and/or hand-coded programs. These program segments are linked to each other.

Floating jump addresses corresponding to IF or GO TO statements, or to the statements calling for subprograms, and to the references to temporary storages, are replaced by the actual ones. This replacement is accomplished with list-structured tables directly mapped on the program itself.

Moreover, the loader modifies the absolute codes for the complex arithmetic interpreter.

This loader, on completion of its functions, is overlayed by library functions and input-output control routines.

### 3.3. *Input-output control routines*

An input-output statement is converted to a calling sequence of input-output routines. It includes the location of the format and the list of operands concerned.

Input-output control routines edit output data or translate input data into internal representations as well as control input-output units' actions.

### 3.4. *Subprogram and its auxiliary routines*

A subprogram coding has, as its components,

(1) jump to (6),
(2) input-output formats,
(3) jump instructions corresponding to Assigned GO TO statements,
(4) arithmetic statement functions,
(5) the coding body,
(6) the linkage to the auxiliary routine which replaces dummy arguments by the corresponding actual ones,
(7) the calling sequence for (6),
(8) jump to (5),
(9) temporary storages.

## 4. *Complex Arithmetic*

### 4.1. *Imbedding of complex arithmetic*

An acceptable complex number may be restricted to that of floating type.

It is represented by a pair of its real and imaginary parts.

With respect to the arrangement of the pair, the real parts and their counterparts are grouped into respective areas with definite difference of address.

Library functions such as LOG, EXP, SIN, COS etc. are redefined. Built-in function ABS, rather complicated, cannot but be a closed subroutine.

Some operators newly added, peculiar to complex arithmetic, are as follows.

(1) R (complex variable)=arithmetic expression,
(2) I (complex variable)=arithmetic expression.

These two evaluate the expression on the right hand side and assign it to the real or imaginary part of a complex variable, respectively.

(3) R (complex arithmetic expression),
(4) I (complex arithmetic expression),
(5) ⓐ (complex arithmetic expression).

These three, in an expression, mean the real part, the imaginary part or the conjugate of a complex arithmetic expresion, respectively.

Defining the imaginary unit with the first two operators (see the example), one can represent a constant complex number.

As to input statements, a complex number is treated as if being a pair of contiguous real numbers and format designations of real part and imaginary one are both put in the FORMAT.

A complex variable must be declared COMPLEX, and a statement having them as its operands has to be headed with K-punch. On acceptance of a K-headed statement, the compiler parenthesizes the statement between the jump instruction to the complex arithmetic interpreter and the pseudo-instruction to put an end to interpretation. The statement itself can be compiled in the same way as non K-headed one with the exception of operators peculiar to complex arithmetic.

### 4.2. *Complex arithmetic interpreter*

The interpreter can be considered to simulate a kind of virtual computer which performs complex arithmetic with some pseudo-registers. In spite of its logically simple structure, for the complicated housekeepings, as a whole, it occupies about 700 machine words including library functions.

### 4.3. *An example*

```
C  INTEGRATION OF 1/Z ALONG A SQUARE AROUND ORIGIN.
     COMPLEX A, DELTAZ, SQINT, UI
K  R(UI)=0.
```

```
K       I(UI)=1.
K       SQINT=0.
K       Z=1.
K       DELTAZ=-.01+UI*.01
        DO 100 I=1,4
        DO  10 J=1,100
K       SQINT=SQINT+DELTAZ/Z
K 10    Z=Z+DELTAZ
K 100   DELTAZ=DELTAZ*UI
K       PRINT 1000, SQINT
  1000  FORMAT (16H SQUARE INTEGRAL 2F 15.5)
        END (2, 2, 2, 2, 2)
```

The second line of the program declares Z, DELTAZ, SQINT and UI to be complex variables. The seventh assigns $-.01+.01i$ to DELTAZ.

## 5. *Supervisory Program*

On completion of compilation of a (sub)program, if the ASW1 is UP, the loading and execution phases are entered. The supervisory program controls these phases, that is,

(1) reads in the loader and transfers control to it,
(2) takes necessary actions when a supervisory information is read in,
(3) executes the object program,
(4) repeats loading and execution or proceeds to further compilation according to the given supervisory information.

At present the function of the supervisory program is limited to loading and execution. But it will be expanded over the whole system in the modified version (model 813).

*Acknowledgements*

This compiling system has been operational at the Computing Center of the Tokyo Institute of Technology since July 1963.

The authors are thankful to the members of the Institute, especially to Assistant Professor H. Kobayashi, for suggesting many corrections and improvements.

They should like to thank, also, K. Inoue, Lecturer of the University of Tokyo, and their senior collegues, Mr. D. Kobayashi, Mr. H. Matsubara, Mr. T. Ikeda and Mr. M. Yamazaki for kind encouragements, and Mr. I. Saoda for thorough reading of this paper and many linguistic improvements.

Thanks to their collegues, Mr. Y. Konno, Miss R. Komatsu, Mr. S. Minami, who were once (and/or are) coworkers and contributed (and/or are contributing) to this system.

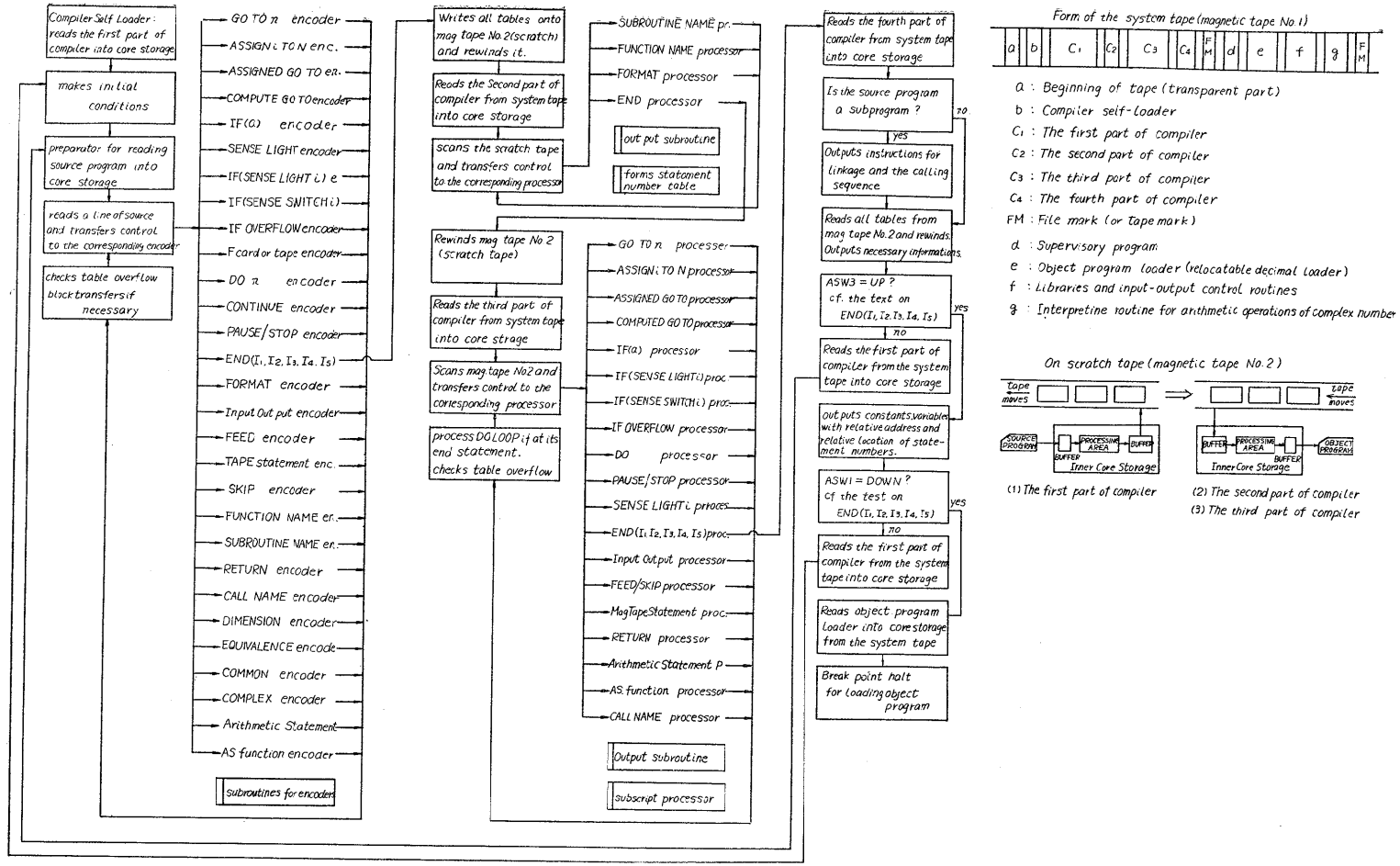


Fig. 1. General Flow Chart of FAST Compiler