MASSACHUSETTS INSTITUTE OF TECHNOLOGY

2/3/73 19

Memo to Jerry Room Ext.

Thought you might want to hold onto this. It is a collection of documents concerning the partially implemented Dynamo compiler that Martin Jack worked on.

Vic

from Room Ext.

This term I have transferred the existing 360 PL/I code to Multics. This necessitated:

Coding and testing of dynamo_operators_.
Rearranging dyn_c_error_ to output messages directly to user_output, and elimination of the procedure edump.
Transferring dyn_trans_ nearly as-is.
Transferring dyn_sort_ nearly as-is.
Transferring dyn_sdump_ nearly as-is, but changing the method by which the equation numbers were converted.
Transferring dyn_ddump_ as is, and fixing a serious bug.
Recoding dyn_lex_ completely, incorporating ctlproc.
Rearranging dyn_codegen_, and redesigning the object code macros.

In addition, the structure of the data bases was completely rearranged from all static external with refer-based structures to based structures overlaid on working segments.

The object code interfaces and the interfaces with the operating system had to be completely redesigned to fit in with Multics.

What needs to be done to get it working

There is some kind of minor bug in dyn_sdump_ with the cross reference numbers. I think what is happening is that the rightmost digit is being cut off. fixed.

The remaining undebugged code: dynamo_operators_

The plot routine is in skeleton form right now so that a raw dump of the v_tabs can be obtained in order to be sure that they are right before proceeding with a plot routine.

Improvements to the implementation

A macro processor needs to be added, in order to facilitate the implementation of the built-in functions delay3 and clip, which use temporary variables which ought to be generated via a macro call. Also, the table facility should be added as soon as possible, since people use them widely.

It would be hard to say what could be done to the implementation. So much depends upon the reordering phase that a major change would be hard to make. It might be nice to make the symbol table into a hash table, but it has to be linearly scanned so much that that might not be such a good improvement. I think the thing is about as efficient in syntax and godegeneration as it can get. Some minor bit fiddling could be done in reordering but I don't recommend doing anything to it as it is very obscure in places and in fact there is only one person in the world who completely understands how it works, namely JRN, who wrote it. ?

dyn_sdump_ was fixed to char conversion
pw1 & pw2 are currently external static.
some modules (e.g. dyn_trans_) misdeclare labels so unwinder is invoked.

The following modules are part of the dynamo compiler:

dynamo_operators_
: Performs run-time services for the object program which are unwieldy when coded directly in machine language. There are currently two entries: dyninit_, which initializes the stack frame for the object program from the run-table chains, and dynio_ which interfaces to the data collection tables and which is called from the object program when it needs to have the data collection tables updated.

dyn_e_error_
: Issues diagnostics to the user's console when errors are detected during compilation.

dyn_trans_
: Performs syntactic and semantic analysis of the source program. It generates the intermediate code chains. It is a stack-driven bounded context analysis program (operator precedence). It performs evaluation of constant expressions.

dyn_sort_
: This phase reorders the intermediate language according to the cross-reference bits into the order in which the code must be generated.

dyn_sdump_
: Generates a formatted listing into the listing segment of the identifiers used in the compilation with attributes and references.

dyn_odump_
: Generates a formatted listing into the listing segment of the state of the ordering table after reordering. This program is primarily a debugging tool.

dyn_lex_
: This is the lexical analysis routine for dynamo. It is responsible for generating a source listing if requested, parsing the input segment into tokens, returning them one at a time to syntactic analysis, and processing control statements"$...;". There is an initializing entry setup_, which sets up internal static variables for the main routine

dyn_codegen_
: This is the code generator. It uses the intermediate code tree as input and produces GE645 machine code into the object segment.

dyn_alm_
: This is an alm program used as a transfer vector to obtain external linkage from the dynamo object code. The function of this program is described more completely where the object program environment is described.

dyn_odump_

This rather simple procedure simply uses the ordering table work_segment_2.order and the indexing variables n_l,n_s,n_a,n_r, and n_i to format a listing of the order in which the equations will be compiled to the listing segment.

dyn_sdump_

This procedure produces a formatted listing of the symbol table. It first uses the array deqn as a temporary area to sort the symbol table (maintaining indexes to sym_tab in deqn) via a simple interchange sort.

It then loops over all entries in the symbol table (that is, deqn) to print the attribute and cross reference listing.

The first line contains the information from sym_tab, that is, a '*' for inflag, and the def and init pointers, the ASCII name of the symbol, and the type of the symbol (translated to a character string via sym_type).

dynamo

This is the command module, entered from the command processor. It is basically a dispatching module, calling the various phases of the compiler based on the options specified. It also processes the command argument list, creates and truncates the working segments, and initializes the working variables in the common data segments.

Data Bases

In general, each program has internal automatic variables which are completely local in nature, and in general few in number. Most of the data common to all phases of the compiler is held in two working segments which are declared by the include files dynamo_1 and dynamo_2, and are created by dynamo and the static external pointers pw1 and pw2 set to point to the created segments.

The first work segment contains element variables which are global in scope, and a large area in which allocation of free storage is done. The second contains an interleaved array in which all of the data which is array-structured is kept. The reason that the array was interleaved is to allow the working set of this segment to vary uniformly with the sizeof the input program.

Description of dynamo_1.incl.pl1

Work_segment_1 is overlaid on ¢ pd¢ dynamo_temp1_.

n_dev Number of device interfaces to be generated by codegenerator. This number is currently fixed at 2 but may be made variable at some future time.

n_cst Number of entries in the constant table, work_segment_2.c_tab.

n_eqn Number of equations in the model.

n_sym Number of entries in the symbol table, work_segment_2.sym_tab .

n_var Number of allocated entries in the cross reference bit arrays, work_segment_2.crossv and work_segment_2.crosse .

n_err Number of entries which have been made in the (new defunct) error table. This field is not currently in use.

sp Stack pointer (used only by lex_ and trans_).

tsp Stack pointer to tstack (used only by lex_ and trans_).

line_no Current source program line number.

l_run lex_'s opinion of the current run number. Used for synchronization.

t_eqn trans_'s opinion of the equation number.

t_run trans_'s opinion of the run number.

cc_code Maximum severity code of an error detected so far.

n_a
n_[illegible]
n_l
n_r
n_s These five variables are all indexes into work_segment_2.order and are used to indicate what order equations are to be generated in. See the picture of the ordering table somewhere else to see what these variables are used for.

m_comp Extent of the ordering table, work_segment_2.order .

first_run Set to indicate that the first run is in progress. This is an indicator to dynamo_operators_$dyninit_.

lhs Set to indicate that parse is on the left side of an equation. Used only by lex_ and trans_.

lnext Next available character in the listing segment.

lseg Length (in characters) of the source segment.

psource Pointer to the source segment.

pobject Pointer to the object segment.

pentry Was supposed to have been a pointer to the entry point of the object segment. Is not being used now since same as pobject.

pstack Pointer to the stack (which is in trans_ automatic storage). So that lex_ can find it.

ptstack Same as for pstack, but for the error recovery stack tstack.

plist Pointer to the listing segment.

pr Pointer to the run table chain. See the picture of the run tables.

rptr Pointer to the current run table entry.

options The compiler options in use.

crav Used by the reordering phase.

area_a Used for allocation of based variables.

Description of dynamo_2.incl.pl1

Work_segment_2 is overlaid on ϕ pdϕ dynamo_temp2_.

c_tab Table of all constants encountered in the dynamo source program and the rsult of evaluation of all constant expressions. The variable n_cst contains the maximum index used to date.

sym_tab The symbol table. Fields are used as follows:

name The ASCII symbol.

iflag Set to one when the variable appears in the initialization chain from r_tab.rvars.

type The type of the symbol. Codes are as follows:

0	constant
1	auxiliary
2	rate
3	function
4	macro (not implemented)
5	level
6	supplementary
-2	encountered in a $dev, and entered, but not encountered in the program, so no data type can be assigned.

cross Index in the reordering cross arrays for the variable.

def Number of the equation where defined (until code generation)
Offset within the stack frame (during code generation)
Index of the value for a constant variable (during execution)

init Number of the equation where initialized.

crossv Reofdering cross array, by variable (cross field of sym_tab).

crosse Reordering cross array, by equation number (n_eqn).

deqn Used in reordering.

eqns Equation table. Usage: ← ?

dsym Used in reordering.

dvar Used in reordering.

eq_tab Pointer to the intermediate text blocks for the [redacted].equation.

order Indexes in eq_tab for the result of the reordering process. The variables n_a,n_i,n_r,n_s,n_l give indexes in order for the different variable types. See the picture.

[redacted]Description of dynamo_m.incl.pl1

Formats of the intermediate text blocks.

MATRIX (A line with two arguments)

chain pointer to next block or null.

op Operation code for this line

t1 Type of operand 1 (sym_tab.type field)

op1 See 1.2.

Description of module dynamo

First off, we output a message to the user's console indicating that we are working. Then, all of the options given are processed. We make sure that at least one argument (pathname of source program) exists and then process the 2nd through nth arguments, trying to match them against the valid options in arg_tab. For each one which was given, we set the corresponding bit in arg_bit. For options not recognizable, we call the error handler to issue diagnostics. Then, the three temporary segments are made by a call to make_seg.

We then process the pathname of the input program. We check for the suffix .dyn and add it if necessary, and then call expand_path_ to get the full pathname. Then, we initiate the segment, and set the appropriate variables in work segment 1. The remaining variables in the two working segments are then initialized. The first r_tab is allocated and initialized. If any options requiring the presence of a listing segment have been given, then a listing segment is created in the working directory.

After this point, a simple set of calls are made to the appropriate modules of the compiler based on the success of compilation as measured by cc_code, and the option flags. When compilation and execution are complete, the code following the label quit terminates the source segment, sets the bit count on the listing segment and terminates it, and then truncates the working segments.

Description of module dyn_alm_

"(see description of the procedure dyn_codegen_)"

This module is the means whereby the dynamo object program is able to make external references. The second word of the object program is initialized with a pointer obtained via make_ptr to the entry dyn_alm_$dyn_tv_, and the following calling sequence is used to reference an external entry:

```
lxl5  offset,dl     load the offset in transfer vector
stcd  sp/20         save return point
tra   2,*           transfer indirect through saved pointer
                    to transfer vector
```

Within the transfer vector, the first instruction transfers to location 1 indexed by the contents of index register 5, thereby accessing either an external symbol or an alm subroutine coded in dyn_alm_. These subroutines had not been coded as of May 1971.] ← status now?

Description of module dyn_trans_

This module performs syntactic and semantic analysis. It uses an operator precedence analysis technique to parse the input stream.

The top of loop is at bca, where the equation counter is incremented, and a left pad is put on the stack. One token is lexed, and checked for identifier. If not, then a syntax error exists. The next token is check for "=". If not, then one is inserted. Then, precedence analysis is used to determine where the reducible phrases are.

Semantics for constant operator constant are to evaluate the constant expression, placing the result into the next available spot in c_tab, and reduce the source. When a constant_variable = constant is recognized, an rv_tab is allocated and added to the chain off r_tab.rvars, and initialized so that the constant variable will be initialized at run time by dyninit_.

For arithmetic operators, the routine iop allocates and chains a binary matrix line block and fills in the appropriate fields. If the operator is assignment, then end of statement has been reached and cleanup occurs.

Unary minus operations are accomplished by adding a unary minus matrix block or in the case of a constant placing the negative of the constant indicated into the next c_tab location.

Exponentiation is handled as a constant evaluation if possible, or a function-call matrix is allocated, and initialized. The name of the function invoked is "**exp". For function calls, the appropriate function call matrix line is allocated, and initialized.

When the stack is to be cleared after the end of a statement, the variables t_run, t_eqn, and l_run are examined and set for synchronization with lex regarding end of statement.

Description of module dyn_lex_

Dyn_lex_ has the function of returning one lexical token on the syntactic analysis stack when called at entry point dyn_lex_. It must be initialized via a call to dyn_lex_$setup_ first. It will also process control statements in general by modifying the run table chains.

A lexical token comprises a token, type, and name cell. The following codes are used:

token = 0 ** 1 * 2 / 3 + 4 -
5 = 6 un- 7 , 8 9 ;
10 (11) 12 13 idn

type (for token=13)
0 .j 1 .k 2 .jk 3 .kl
4 nosub 5 const 6 matrx

name (for token=13)
for type=0,1,2,3,4 - index in sym_tab
type=5 index in c_tab
type=6 matrix line number

(The token=6,8,12 and type=6 indications are set only by dyn_trans_.)

The internal procedure endline is called when a newline character is encountered. It increments the line counter, and outputs the line just completed to the listing if a source listing has been requested.

The internal procedure nextokn returns the next token to the caller (either lex or ctlproc) as nptr, nmod, and ntyp. The rules are as follows: if the bit eof is on, then return. Skip all blank, tab, and newline characters before the next significant character. Set nptr to point to the significant character. If this character is numeric, then the token is a numeric item. If a special character, then the token is an operator. Otherwise, it is an alphabetic item.

For operators, the special cases "**", ":=", and comments are recognized. The handling of each of these is relatively straightforward. For single character operators, nmod will have already been set to the token field above.

Numeric items are delimited on the right by a special character, blank, tab, or newline. Alphabetic tokens have the same restrictions, except that the time subscript has to be checked for and returned as nmod.

dyn_trans_

The mainline code checks if anything has been placed on the temporary stack (trans_ can place tokens on this stack if it wants to insert a token for error handling.) If so, then this token is returned. If a $run; statement has been encountered, a right pad is returned, or if physical end of data on the segment has been reached, a right pad is also returned. Otherwise, a call is made to nextokn to get another token, and the processing depends upon the type (operator, numeric literal, identifier).

For operators, the flag lhs (left hand side) is set or reset if an "=" or ";" is reached. If the "$" character occurs, then ctlproc is entered

to process the control statement which follows. If the construction ":=" occurs, this is returned as token=5,type=1, to distinguish from "=", which is returned as token=5,type=0.

For literals, the character string token is converted to floating point and stored in the next available c_tab position, and returned as token=13, type=5,name=index in c_tab.

For identifiers, the symbol table must be searched to determine if the identifier already was recognized. If so, then token=13,type=some function of the time subscript given by tp_tab, and name=symtab index. If the identifier must be entered, then the symbol table entry must be initialized, and a slot allocated in the crosse reordering array. The pointer n_var is updated to indicate the allocation, and the index in crosse(*) is stored in sym_tab.cross. The cross arrays are updated to indicate usage on the right and left hand sides, and multiple declaration and initialization are checked for. The token is then added to the stack.

The routine ctlproc processes all control statements. The variable ntoken will contain the verb (run,dev,spec).

For a $run statement, a check is made to be sure that the dt and length fields are filled in for the previous r_tab, and that some $dev statements were supplied, and then a new r_tab is allocated chained and initialized. The input stream is then flushed to the next semicolon.

For a $spec dt=,length=; statement, the routine kw_parse is used to obtain the parse of the keyword expression and then r_tab.dt and r_tab.length are filled in with the user's values.

For a $dev unit= outper= idn= statement, a dv_tab is allocated and added to the device chain for the run. For each identifier specified, a pv_tab is allocated and chained to the newly allocated dv_tab. If an identifier which has not yet been encountered is used in a $dev statement, it is given a type of -2 so that a type conflict diagnostic will not be produced, and a contextual type can be assigned later when more information is available. The dv_tab is not chained to the r_tab until all parameters on the statement have been scanned so in the event of an error no effect will remain.

The routine kw_parse uses an lr(0) parse (DeRemer's thesis PhD MIT) using an encoded finite state machine to parse the keyword control statements. Its operation is rather straightforward.

Description of module dyn_sort_

This module uses the reordering cross arrays crosse and crossv and the array order(*) to produce anm indication of which equations are dependent upon one another and therefore resolve the order in which the equations should be compiled. The groups are taken in the order supplementary, auxiliary, initial value, levels and rates. The operation of this module is extremely complex and it is recommended that the author, John R. Nestor, be contacted if modifications are to be made.

Description of module dynamo_operators_

This module performs run-time services for the dynamo program.

The entry dyninit_ initializes the object program's stack frame.

The length cell is set to minus the number of iterations over the model so that a test can be made by the object program via aos-tnz whether enough iterations have been made. The cells variables(54) and (55) contain the values of dt and time. These values are obtained from r_tab, for the current run.

For the first run of a group, the constant variables all have to be initialized, and the sym_tab.init cell is set to the c_tab index for the appropriate value. If not the first run, only the non-permanent constants are initialized from the rv_tab chain; the others are initialized from the sym_tab.init field, so that permanent constants can be retained. For permanent constants, the sym_tab.init field is reset so that subsequent reruns will obtain the new value of the constant.

Finally, the v_tabs for each of the variables to be output are allocated and attached to the appropriate pv_tab.

The entry dynio_ collects the values of variables for later printing or plotting. It loops over all devices active that is dv_tabs, and for each pv_tab extracts the value of the associated variable (the index within variables(*) is contained in the sym_tab.def field) and enters in the v_tab.

Description of module dyn_codegen_

This module is the dynamo code generator for Multics. It generates GE645 machine language directly into the object segment. The object segment is not in the standard Multics format, but rather depends upon support code within the compiler for execution time services and its environment. The input to the code generator is the intermediate language tree constructed by the semantic translator and as reordered by the reordering phase.

Some portions of code generated by the code generator are fixed in nature, and are stored in arrays of static initial bit strings and copied into the object segment with some minor filling in of offsets, numbers, and so forth. Code which is generated for the matrix lines is in general more variable and is generated by encoded macros.

Storage allocation is the first task performed by the code generator. All storage for variables declared in the dynamo program is held in the stack. Some storage is obtain after the initial 32 word header for internal compiler and run-time variables, and then programmer variables are allocated beginning at offset 54 from the sp.) For level and rate type variables, two words are allocated, and for ordinary variables, one word is allocated. No particular alignment of the two word allocations is observed.

Next, the working variables lbound(*),hbound(*), and incr(*) are initialized to indicate what areas of eq_tab will be examined and in what order. The general form of the object code is shown in fig. 1.

The procedure prologue code is generated first. The variable loc contains the index within object_bits (which is overlaid on dynamo_object_) which is currently the first unused location. The procedure prologue adjusts the sp (performs a standard Multics save sequence), stores the current value of sp in words 6 and 7 of the object code for dynamo_operators_, establishes the value of bp by loading from word 4 and 5 of the object code where a pointer to dynamo_temp2_ has been stored, and performs the first part of the call sequence. It then calls the 0th entry in the transfer vector, which is dynamo_operators_$dyninit_, which completes the initialization of the object programs stack frame via reference to the run table information.

Next, code is gnerated for the initial value equations which contain expressions. This is done in the main loops. There are two nested loops. One goes from 1 to 5 and references the lbound and hbound arrays; the other goes from lbound(i) to hbound(i). Within that loop, one simply chains from eq_tab(*) while the chain pointer is not null.

Following initial value equations, a short piece of code is generated to transfer around the calculations which follow and to re-enter just before auxiliary equations are generated. This branch, being forward, is fixed up after the level equations are generated. Then, auxiliaries are generated, followed by a piece of code to test for a print or plot time.

There is one word in the stack at offset 35 which contains flags used by the print/plot data collection interface dynio_. Bits 16 and 17 of this word correspond to devices 1 and 2 and when set indicate that the "outper" for this device has expired and that data should be collected on the current call to dynio_ for all pvars chained from that dv_tab. The outper is maintained in words 36 and 37 of the stack and they are set such that an aos on each loop will cause decrementing to zero when the interval next expires. The code generated after rate equations does aos on these words, tests for zero, and if zero sets the appropriate bit in word 35. Finally, the supplementary equations are compiled, and a test is made of the device flags in word 35 to determine if either of them is on, thereby indicating that data is to be collected for some device on the current loop. If so, a call is made to dynamo_operators_$dynio_ to collect the data. Next, the generated code in epilogue_code adds dt to time to get the new value of time, swaps the .j and .k index registers, inverts the jk-bit in word 35 (this bit is used by dynio_ to figure out if .j or .k comes first in the two-word level and rate slots), does an aos on word 34 which contains the negativeof the number of loops over the model equations to be made, and if this word has not been incremented to zero, branches back to the level equations to begin another loop. If it is zero, then the standard Multics return is made.

The processing performed for generation of code for matrix lines is now described. First, the opcode is extracted and assigned to the variable op. If zero, then a function matrix line has been encountered. First, a temporary is obtained to store the result in (rt). tc(rt) is set to indicate that temporary rt contains the result of the current matrix line. Then, the accumulator is stored into a temporary if rc is not zero, so that it can be preserved over the call. Next, the name of the function is looked up in func_tab, and the index is saved in fx for the lxl5 instruction. Next, if the location counter is not even, a nop 0,du instruction is generated for alignment. Next, a loop is made over all arguments to the function. A search is made for the location of the argument in storage and an appropriate address halfword and modifier bits are constructed (description below). An eapap and stpap instruction are generated to get the address into the parameter list for each one. Following that, an eapap and stpap get the address of the result temporary into the parameter list. Next, an eapap instruction loads the address of the parameter list into the ap, and a tra branches around it. Following the parameter list, an stcd and tra 2,* get through the transfer vector to the desired function. This completes processing for function calls.

If not a function matrix line, then there is at least one operand. The first operand is located by noticing what the t1 field of the matrix line contains. If a 6, then it is a matrix result, and is either in the a-register or in a temporary. The bit op1nir is set if op1 is not in a register. The op1addr is set to the offset in the stack of the temporary where it is stored, and op1modifier is set to zero so that no address modification will be performed. Op1temp is set to the numberof the temporary so that it can be released at the end of generation of this matrix line (it cannot be released now; in the event another temporary is required during the generation, it would overlay the data).

If a constant (type 5), then the offset from the object-time contents of the bp is determined via the rel function, and op1addr is set. Op1modifier is set to zeros, and op1nir is set to indicate not in a register. Otherwise, the operand is a programmer variable, and the field sym_tab.def will have been set by the storage allocator to the offset of the allocation within the stack. Op1addr is set to this address, and op1modifier is set to the appropriate address modifier (0 for an ordinary, x1 for a .j, and x2 for a .k). The array (typ_mod) makes this conversion.

If the matrix line is a unary minus, then the operand is loaded if not in a register, and the fneg instruction generated to reverse the sign.

If not a uminus, then the second operand is foundas for the first, and the corresponding op2... variables set.

For an assignment matrix block, if the operand is not in a register, it is loaded, and then a store instruction is generated to store the data intothe new assignment.

For arithmetic operations, the machine instruction generation depends upon whether the operands are in registers and whether the operation is a commutative one.

For neither operand in a register, a (store) load operate group is generated. The array (arithops) contains the opcodes for the four instructions (fad, fsb, fmp, fdiv). For one operand in a register, if the correct one, or if the operation is commutative, then the correct operation is generated. If not the correct one, then a store load operator group is generated. XXX The variable rc is set to indicate that the AC contains the result of the current matrix line, and the next line is processed.

Cleanup for the current line consists of resetting flags, and freeing temporaries where indicated by the opxtemp settings.

INTERMEDIATE LANGUAGE

The intermediate language tree is a series of threaded lists, one per equation, chained from eq_tab(*). The array order(*) indicates in what order the equations are to be compiled, as given in the diagram.

There are three formats of intermediate language line blocks. One is used when a binary operator is indicated:

```
dcl 1 matrix based,
      2 chain ptr,
      2 op fixed bin,
      2 t1 fixed bin,
      2 op1 fixed bin,
      2 o2 fixed bin,
      2 op2 fixed bin;
```

[illegible]

A second is used when a unary operator is indicated:

```
dcl 1 matrix1 based,
      2 chain ptr,
      2 op fixed bin,
      2 t1 fixed bin,
      2 op1 fixed bin;
```

A third is used for function invocations:

```
dcl 1 fmatrix based,
      2 chain ptr,
      2 op fixed bin,
      2 name fixed bin,
      2 n_args fixed bin,
      2 arglist (i refer (n_args)),
        3 type fixed bin,
        3 narg fixed bin;
```

The type and narg fields correspond to the t and op fields of the previous blocks. The name field is a unique identifier assigned to a function name, and is the index of the name in the symbol table.

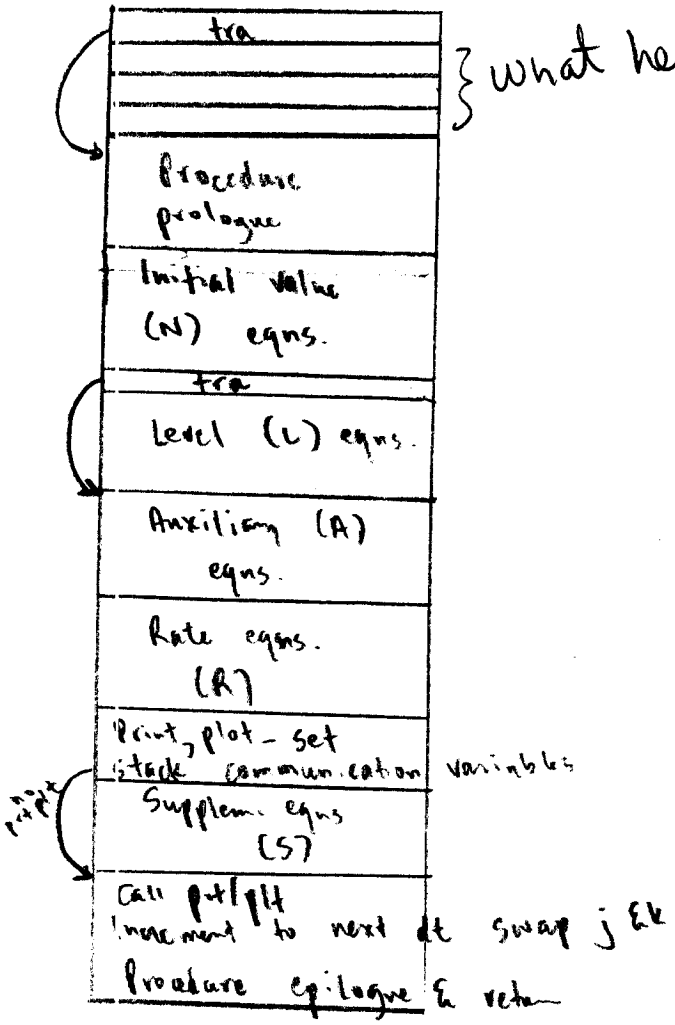
The op field is as follows:

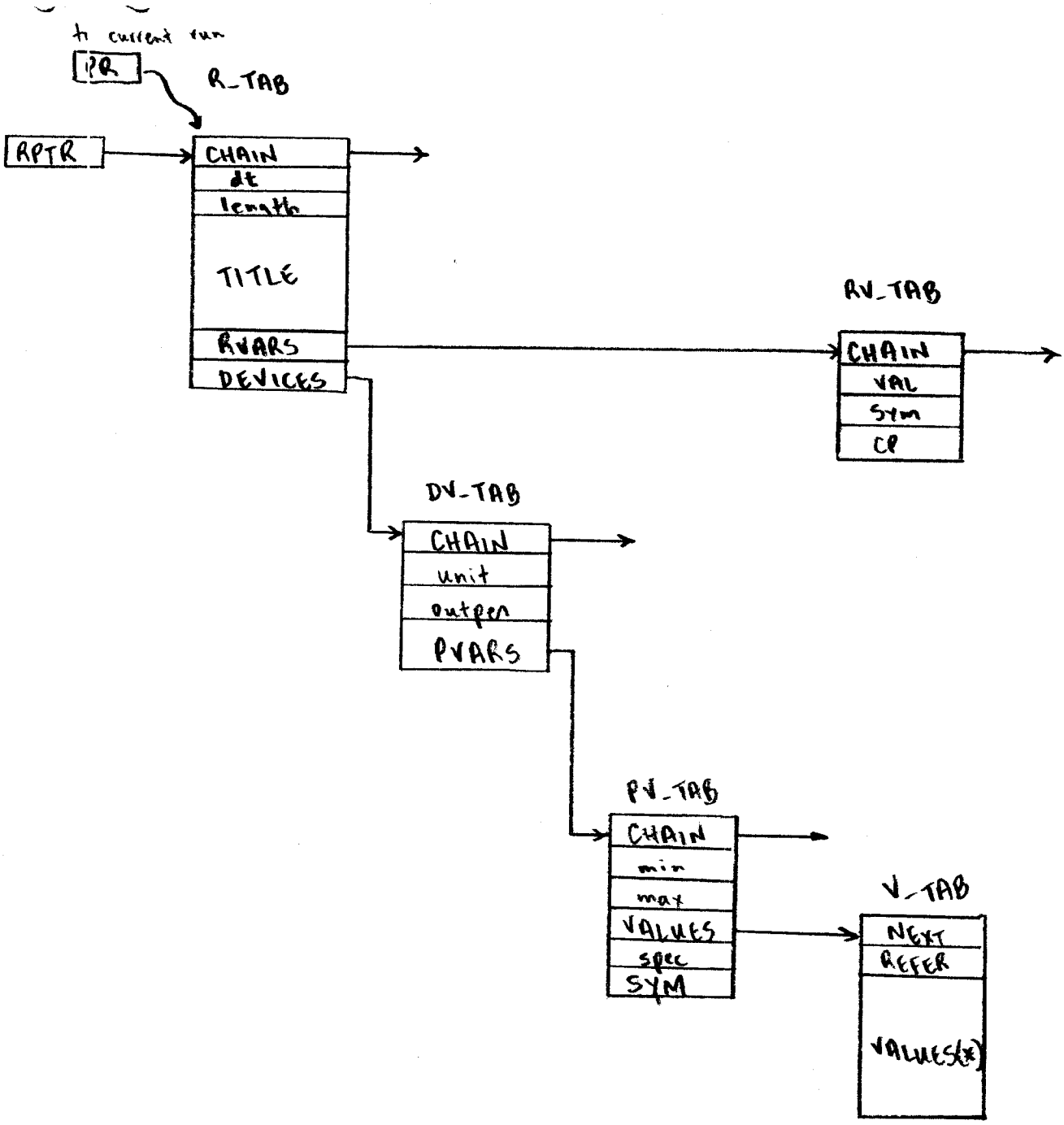
0	function
1	*
2	/
3	+
4	-
5	=
6	unary -

The type fields are as follows, with the operand field corresponding:

type = 0	.j	name = index in symtab
1	.k	"
2	.jk	"
3	.kl	"

4	ordinary	"
5	constant (literal)	index in c_tab
6	matrix result	line number in this equation.


tra
} what here?
Procedure prologue
Initial value (N) eqns.
tra
Level (L) eqns.
Auxiliary (A) eqns.
Rate eqns. (R)
Print, plot - set
stack communication variables
Supplem. eqns (S)
Call put/plt
Increment to next dt swap j & k
Procedure epilogue & return


th current run
IPR
R_TAB
RPTR
CHAIN
dt
length
TITLE
RVARS
DEVICES
RV_TAB
CHAIN
VAL
SYM
CP
DV_TAB
CHAIN
unit
outpen
PVARS
PV_TAB
CHAIN
min
max
VALUES
spec
SYM
V_TAB
NEXT
REFER
VALUES(*)

Harris

I Operator Codes

CODE	STACK-LEX	STACK-TRANS	MATRIX Operators
0	**	**	Function Call
1	*	*	*
2	/	/	/
3	+	+	+
4	-	-	-
5	=	=	=
6		- unary	- unary
7	,	,	
8		+ SOS	
9	;	;	
10	(	(	
11	)	)	
12	EOF	EOF	
13	NAME-NUMBER	NAME-NUMBER-MAT	

II SYMBOL TABLE

```
DCL  1 SYM_TAB(M#SYM ) CTL,
       2 NAME    CHAR (32),   IFLAG BIT(1),
       2 TYPE    FIXED BIN(15),
       2 CROSS   FIXED BIN(15),
       2 OFF     FIXED BIN(15),
       2 INIT    FIXED BIN(15) ;
```

INDEX → DCL #SYM FIXED BIN(15) STATIC;

CROSS - Index in CROSSE

OFFSET FROM CROSS

0 - ORDINARY, .0 .00

1 - .J .JK

2 - .K .KL

OFF - EQN # defined in or 0 - undefined

INIT - EQN # initialized in or 0 - uninitialized

IFLAG - VALUE CAN BE CHANGED FOR RERUNS ('1'B)

TYPE 0- ORDINARY

1- Level TYPE AUXILLARIES

2- RATE

5 in Trans → 3- Function

4- MACRO

5- ~~LEVEL~~

6- ~~Supplementary~~

14

↑ LEVEL

Supplementary

↑ SET IN SORT

Set in Lex

III MATRIX

```
DCL 1 MATRIX BASED(MPTR),
      2 CHAIN PTR,
      2 OP FIXED BIN(15),
      2 T1 FIXED BIN(15),
      2 OP1 FIXED BIN(15),
      2 T2 FIXED BIN(15),
      2 OP2 FIXED BIN(15);
```

Unary –

```
DCL 1 MATRIX1 BASED(MPTR),
      2 CHAIN PTR,
      2 OP FIXED BIN(15),
      2 T1 FIXED BIN(15),
      2 OP1 FIXED BIN(15);
```

FUNCTIONS

```
DCL 1 FMATRIX BASED(MPTR),
      2 CHAIN PTR,
      2 OP FIXED BIN(15),
      2 NAME FIXED BIN(15),
      2 #ARGS FIXED BIN,
      2 ARGLIST(N REFER(#ARGS)),
        3 TYPE FIXED BIN(15),
        3 NARG FIXED BIN(15);
```

```
DCL AREAM AREA(AMSIZE) CTL;

DCL C PTR BASED(CHAIN1);
```

OP - See PART I

NAME - Index in SYM_TAB of FUNCTION NAME

#ARGS - NUMBER of ARGUMENTS

STACK_TYPE TYPE, T1, T2		STACK_NAME OP1, OP2, NARG
0	.J	INDEX IN SYM_TAB
1	.K	
2	.JK	
3	.KL	
6	MATRIX RESULT	MATRIX BLOCK #
4	ORDINARY VARIABLE	INDEX IN SYM_TAB
5	CONSTANT	INDEX IN C_TAB

IV STACK

```
DCL   1  STACK (MAXS) CTL,
         2 TOKEN  FIXED BIN(15),
         2 TYPE   FIXED BIN(15),
         2 NAME   FIXED BIN(15);
```

INDEX ⟶ DCL SP FIXED BIN(15) STATIC

TOKEN - SEE PART I

TYPE, NAME - SEE PART III