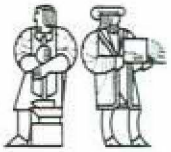
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-18

AUTOMATIC CODE-GENERATION

FROM AN OBJECT-MACHINE DESCRIPTION

Perry L. Miller

October 1970

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

AUTOMATIC CODE-GENERATION

FROM AN OBJECT-MACHINE DESCRIPTION

Technical Memorandum 18

Perry L. Miller

October 1970

PROJECT MAC

Massachusetts Institute of Technology

Cambridge Massachusetts 02139

ACKNOWLEDGMENT

Work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

Perry L. Miller

ABSTRACT:

This memo outlines the basic elements of a macro code-generating system, and develops an informal machine-independent model of a code generator. Then the memo discusses how an implementation of this model could be set up to generate code for a particular machine from machine-dependent information given in descriptive form.

Keywords: compiler, translator, macroprocessor, code-generation, automatic code generation

CONTENTS:

INTRODUCTION:

The compilation process is traditionally divided into parsing and code-generation. A great deal of work has been done in formalizing parsing. Systems have been developed in which a parser can be automatically created from a BNF description of a language, rather than from algorithmic procedure. Very little work, however, has been done to similarly formalize code generation.

A great deal of work has been done on a very closely related problem: that of language transferability. The problem of how to minimize the difficulties of implementing a language operating on a variety of different machines has been approached in several ways. One such approach is typified by the 'mobile programming system' of Orgass and Waite. [6] In this system, the source language is translated into a series of macros by the language processor. Then a user-written set of macros translates this intermediate macro language into user machine code.

A second approach to language transferability is seen in the UNCOL macro language. [8,9] This was an attempt to create a universal macro language into which all high-level languages could be translated, and which itself could be translated into any machine code. If successful, this system would have solved the problem of transferability, since only one translator would ever have to be written for any machine. Notice that this differs from the Orgass and Waite system, since their intermediate macro language was specifically tailored to their source language, whereas UNCOL puts no restriction on the source language at all. In practice, the restrictions imposed by having only one intermediate language have proven very confining and too inefficient for a practical solution.

Both of these systems are similar in that both attempted to solve the problem of language transferability by letting the user specify information about his machine in procedural form. Most of the information about the structure of his machine is buried implicitly in the coding of his macros. This procedural approach has been used in all major published work on code generation.

This memo will describe a system which allows a code generator to be created, for a class of object machines, from descriptive information. In this memo, the code generation process is first formalized, and a machine-independent model of a code generator is informally presented. This model pictures the code generator as:

1. A state machine which makes repeated transitions into permitted states from which it can emit computational machine instructions.
2. Operating on conceptual (semantic) data-types.
3. Built on low-level semantic primitives.

Then the memo shows how an implementation of this machine-independent model could be set up to generate code for a number of different machines from machine-dependent information given in descriptive form.

Chapter 1 first describes the code-generating process in general terms. Then a framework is outlined which allows different data-reference macros and different data-types to intermesh smoothly. This is called the PLOT-LOCATE-LOAD/UPDATE framework. This framework is based on the premise that a data macro need not know whether its result is to be used as an address or a value. As a result, it can compute a '_location_', to which a _load_ or an _update_ function may later be applied. The form of this 'location',

and the particular load or update function used, will be different for the different data-types. Chapter 1 then outlines how this framework could fit into the implementation of a code-generator.

Chapter 2 describes a code generator as a state machine whose state is determined by the location of the values which are to be used in generating code. These values may be in simply addressable core location, in non-simply addressable core locations, or in a register, of which there may be several classes.

Each computational macro has associated with it certain permitted initial states for its operands. For the 360, for instance, initial states for an integer ADD macro would be: 1. both operands in registers, or 2. one in a register and the other in core. The process of generating code for such a macro, therefore, is the process of the code generator making a transition into one of these initial states, followed by the emission of a particular code sequence from that state.

In a procedural macro language, the user specifies how these transitions are to be made. In DMACS, the descriptive macro system of this memo, the code generator itself is set up to perform these transitions automatically. To do this, it must have a description of the register and memory structure of the machine, and of the paths (load, store, register-register transfers) between different storage classes. Thus, computational macros can be defined merely by describing the instruction sequences from permitted states.

Chapter 3 discusses how data-reference macros could be written machine-independently and then filled out by a machine description. In these macros, the user would specify the computations he wanted done, using

'semantic' data-types and semantic primitives defined over these data-types. A machine user could then give a description of his memory hierarchy, its addressability, the operations that move data-items between core and registers, and how his language data-types map into this memory hierarchy. Using this descriptive information, DMACS would map user data-types into the appropriate semantic data-types for that machine, and it would define an implementation for the various semantic primitives over these data-types.

Chapter 4 discusses how further parts of the code-generator could be written in machine independent form and then filled in by descriptive information.

There would be two steps in writing a code-generator using the DMACS system. The first step would be to define a set of procedural macros in a machine-independent, somewhat skeletal form. The second step would be to supply descriptive information about a machine which would be used to flesh out the macro definitions. The two steps would be quite independent, so that once the first step had been done for a given macro language, the second step could then be done for a variety of object machines. To facilitate these two steps, DMACS provides two languages MIML, a procedural machine independent macro language, and OMML, a descriptive object machine macro language. Programs written in these two languages are bound together by the DMACS system.

In summary, the memo is a step towards formalizing the process of code-generation and abstracting it from any particular machine. Toward this end, the memo shows how a code-generator could be created from a machine description. It examines some necessary design features which a

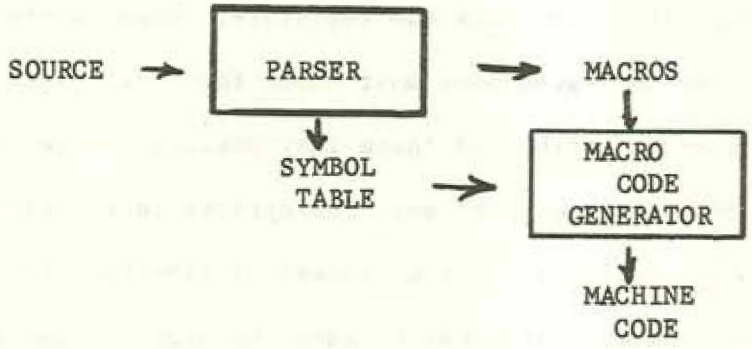
compiler must have to make such machine independence possible. Finally, it describes the BMACS system which is designed to implement such a code generator.

CHAPTER I: THE ELEMENTS OF A CODE GENERATOR

1.1 DESCRIPTION OF A CODE-GENERATING SYSTEM:

A macro code-generator is set up to accept a linear string (sometimes called a 'matrix') of macro calls and generate the appropriate machine instructions. In a complex compiler with many data-types, it is advantageous to allow the code generator direct access to the symbol table constructed in the syntactic pass by the parser. The code-generator can then use the data-type information directly to generate different code to access the differing data-items.

The data flow in such a system is illustrated below:



The parser analyzes the source program, converts it into a linear sequence of macros, while simultaneously building the symbol table. The code-generator accepts the macros and the symbol table as input for generating machine instructions.

The macro-instructions can be thought of as having normal prefix form: for example, ADD X,Y. In fact, this instruction would consist of 3 pointers (also called uniform symbols): (ADD) (X) (Y) Where (ADD) is a pointer into a table of macro definitions, and the operands (X) and (Y)

are pointers into the symbol table constructed by the parser. An operand could also be a pointer to the result of a previously macro line.

Examples of this sort of macro:

	(line no.)	(macro name)	(operands)
A := B+C*D	i	MUL	C,D
	i+1	ADD	i,B
	i+2	ASSG	A,i+1
C := P(N)→A(I).B(J)	i	SS	P,N
	i+1	PTR	i,A
	i+2	SS	i+1,I
	i+3	SUBST	i+2,B
	i+4	SS	i+3,J
	i+5	ASSG	C,i+4

As an example of the code-generation process, consider a simple computational macro such as ADD for the IBM-360. There are two possible add-instructions for integers: 'A' which adds a full word of memory to a register, and 'AR' which adds two registers. When generating code for an ADD macro, the code-generator must check the location of the values to be added to see if either of these instructions can be emitted directly. If not, the code-generator must emit appropriate instructions to load one or both into registers. If in the process of finding a free register to load into, the code-generator has to save the previous contents of that register in a temporary location, this must be recorded. Furthermore, if one of the values is not simply addressable (e.g., a bit string), the code-generator must output appropriate load and shift instructions to isolate it in a register before it can be used in the addition. Finally, the code-generator must record where the result of the addition is located.

To create a code-generator which performs these tasks, therefore, it is necessary to describe:

1. A set of macro definitions.
2. The machine registers, and classes defined over these registers.

3. Routines to keep track of what values are in which registers.

4. A 'GETREG' function to obtain a free register of a given class.

5. Data-handling routines which will generate code to fetch a value into a register, and routines to yield some simple representation of the value, such as an indexed address.

The MPL macro language, which is part of the LPS compiler-building system created by Professor Graham at M.I.T., was developed to offer these facilities. It allows the user to create code-generators for a wide range of object machine structures. The following ADD macro, written in MPL for the 360 is given as an example to give the reader a flavor for such a procedural macro language.

Sample Macro Written in MPL:

```
MACRO     ADD X,Y:
          call  Getloc(X,'REG1,'BASE,'INDEX,'DISP);
          c     REGFLAG,set,t,Xinreg;        (branch if X in register)
          call  Load(Y,'REGL);
          out   (A,REGL,BASE,INDEX,DISP);    (output add from storage)
          callx Mark(REGL);          (set result of current line and exit)
Xinreg:   call  Getloc(Y,'REG2,'BASE,'INDEX,'DISP);
          c     Regflag,set,t,XYinreg;       (branch if Y in register)
          out   (A,REG1,BASE,INDEX,DISP);    (output add from storage)
          callx Mark(REG1);
XYinreg:  out   (AR,REG2,REG1);              (output add register)
          call  Mark(REG2);
          endm;
```

Explanation of logic: Getloc(X,R,B,I,D) is a code-generating subroutine which attempts to return (in the variables B,I, and D) a simply addressable location for X. If it cannot, it sets Regflag and returns a register containing the value in variable R.

Load(Y,R) is a similar routine which loads Y into a register which it returns in variable R.

Mark(R) marks register R as containg the result of the current macro line.

The routines Getloc and Load handle (directly or by subroutine) all the symbol-table searching, all the data-type dependent logic, and any necessary searching for free registers.

Notice that MPL is a procedural rather than a descriptive language and that the user gives a procedural outline of the macro logic. The user must similarly specify procedural logic for handling data-dependent logic and for obtaining free registers.

Chapter 2 will show how computational macros of this sort could be deduced from descriptive information about an object machine, and discusses some of the problems involved. The remaining sections of chapter 1 will discuss further general aspects of a code-generator.

1.2.1 A FRAMEWORK FOR DATA-REFERENCES AND DATA-TYPES:

A powerful computer language, like PL/I, has a variety of constructs for referencing data, including simple arrays, matrices, pointers, and structures. Reference to a single data-item can involve arbitrary compounding of any of these. This gives the user a great deal of power, but presents problems for the compiler writer. This is especially true when these data references are defined not only over words (i.e., addressable units), but also over bytes within words, and bit strings which are not directly addressable at all.

This section describes a framework which allows different data-reference macros and different data-types to intermesh smoothly. This is called the PLOT-LOCATE-LOAD/UPDATE framework. This framework is based on the premise that a data macro need not know whether its result is to be used as an address or as a value. As a result, can it compute a '_location_', to which a _load_ or an _update_ function may later be applied, to either access or alter the value at the 'location'. The form of this 'location', and the particular load or update function used, would be different for the different data-types.

1.2.2 THE 'LOCATE' PROBLEM:

One fundamental problem in generating code for any data-reference is that the isolated reference itself does not indicate whether that data item is to be used as an address to store into, or as a value. Only when it is used in context does this become clear. For example: A(I) := A(I) + B. Here the first A(I) refers to an address, while the second refers to the value at that address.

Similarly, if the subscript was represented by the macro line SS A,I, it would be impossible to determine whether code should be generated to yield a value or an address. One solution to this problem is to let the parser determine the context and output two macros: SSA A,I when an address is wanted, and SSV A,I for a value. A variant of this solution is to let the parser always output an SSA macro, followed by a unary 'value' macro to convert this address to a value when desired. A second solution is to let the data reference macros always compute an address and let the macro which uses this result determine the mode.

A data reference macro which computes an address can be seen as a 'LOCATE' function which generates a representation of the location of the data item.

1.2.3 THE LOAD-UPDATE PROBLEM

The fact that not all data items are simply addressable gives rise to the concept of a load-update pair: a complementary pair of routines to access or update a data-item.

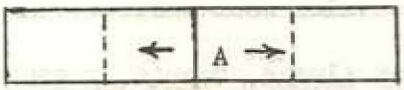
The simplest example of such a 'complex' data item in a word-addressed machine would be a byte within a word. Its 'location', generated by a locate function, might be: 1. an address (possibly indexed), and 2. a byte number. The load/update pair would consist of two routines which would take

this 'location' and generate code as follows:

1. load: load the word into a register, shift left to eliminate high order bytes, then right to eliminate low order bytes, thus right-adjusting the desired byte in the register.
2. update: load the word, use a mask to zero out the target byte, shift the new value to the correct position, 'or' then, store the word with its new byte.

In practice, this data item would have two kinds of 'location' and correspondingly two load/update pairs: one for when the byte within the word is known at compile-time, the second for when the byte within the word is calculated at run time by the locate function, and is given to its load/update pair as a computed value.

The load/update routines would be further complicated if a data item extended across a word boundary,

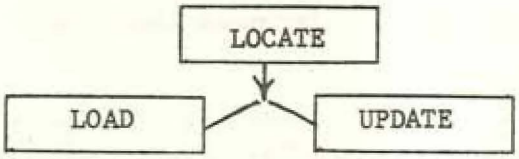


but the code generated would still fit into the framework of a 'location' and load/update pair.

It can be seen that this problem stems from the fact that possible data items do not map directly into addressable units. Generally only an address (perhaps indexed by registers) can be put into a machine instruction. If it were possible to specify an address and a byte number, or address, starting bit, and bit-length, then the problem of special load/update functions would disappear. A machine which allowed this would pay a penalty in efficiency when it was working with full word items. A simpler trade-off might be to have a special hardware load and store instructions to access bits of a word. This would still retain the load/update framework,

but would make the load/update routines much simpler since each would consist of only one instruction.

1.2.4 LOCATE & LOAD/UPDATE:



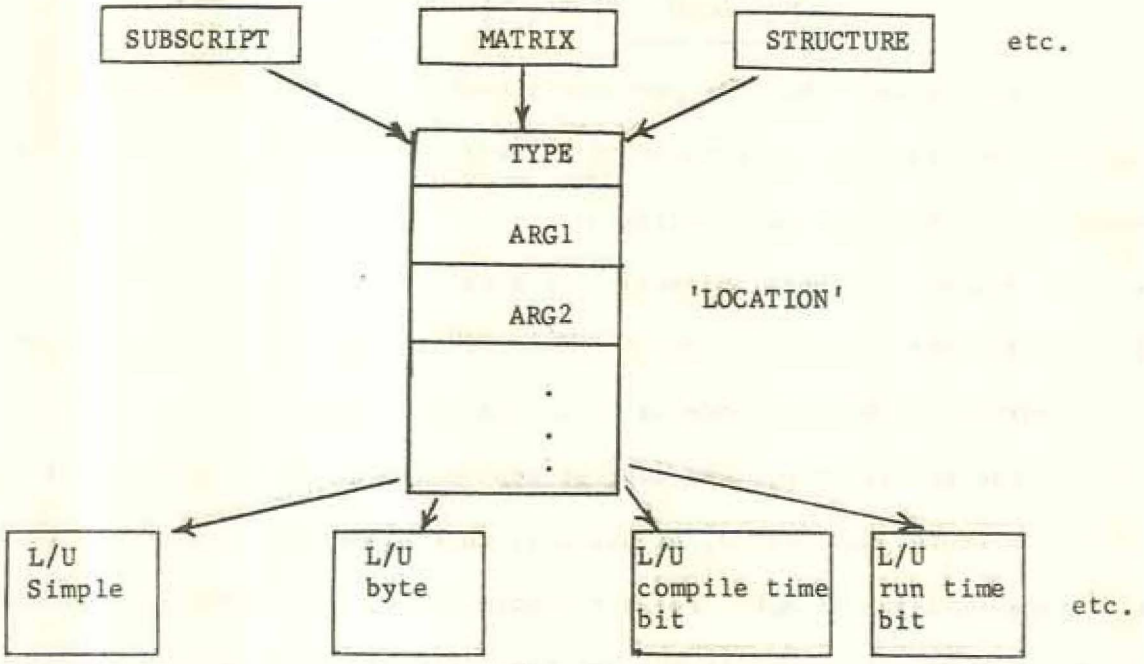
It can be seen that the previously mentioned locate function fits together neatly with the concept of a load/update pair. A locate function generates a 'location' consisting of:

1. An address (perhaps indexed) if a data reference is simple.
2. An address (perhaps indexed) and auxiliary pointers to within that address if the reference is a complex data item.

In the simple case, this address can be fit directly into either a load or a store instruction, or directly into a machine instruction such as a storage-to-register ADD. For non-simple items, the address and pointers can be given as arguments to the appropriate half of a load/update pair, to either generate or value or update the item with a new value.

The various data-structure macros can each perform their own computations on their operands and associate the 'location' thus generated with the current macro line. This 'location' can be any number of run-time and compile-time values, depending on the nature of the data item. When the result of this line is later used, this 'location' will be fed to the proper half of a load/update pair. Since the 'location' format put out for a given type of data item would be the same for all data-reference macros (subscript, matrix, structure, etc.) they could be fed to the same load/update pair. This 'location' format is a type-indicator followed by an arbitrary number of operands. Since the type-indicator tells which load/update applies,

this 'location' can be thought of as a 'twin' function followed by its arguments. Either the load or the update part of this function will be called by a later macro which references this result.



Considering the optimization of common subexpressions helps shed some light on this locate-load/update framework. In optimizing A(B(I)) := A(B(I))+1 the result of the locate of A(B(I)) would be used twice. In optimizing C := A(B(I))+D(A(B(I))) the result of the load would be used twice. Whether these references were simple of complex and no matter how complicated the locate or load/update routines were, or how complicated the 'location', this would remain valid.

1.2.5 A TWO-PART LOCATE FUNCTION:

To handle structures, a modification of this 'locate' idea is helpful. Consider the structure:

A(I).B(J).C(K)

First let us ask how this should look in macro form. One might try as a first attempt to consolidate the whole reference into a single macro:

LOCATE,STRUCTURE A, I, B, J, C, K.

This approach is unwieldy. It hides structural information so that it would be virtually inaccessable to an optimization pass. Also, it would require a proliferation of different macros for various number of operands, and still could never be completely general.

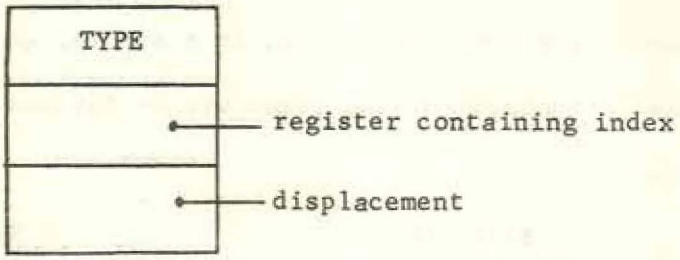
A cleaner approach is to have a 'sub-structure' macro. This would handle <u>any</u> structure format in a simple, general fashion, and would let the 'structure' of the structure be obvious to the optimization pass. For example:

A(I).B(J).C(K)	i	SS	A,I
	i+1	SUBST	i,B
	i+2	SS	i+1,J
	i+3	SUBST	i+2,C
	i+4	SS	i+3,K

There is one major difference between the SS macro as it appears here (in a structure) and as it appears by itself. When A,I appears by itself, the location it generates is passed directly to the load/update routines. When it appears in a structure, however, its result will not be used as a location until the very end of the structure is reached. As a result, there is more flexibility in what the SS macro computes within a structure. Consider the following examples:

Example 1. A(I).B(J).C(K) where all displacements and indices are full words, and array sizes are fixed. Since the displacements of A,B, and C are known at compile time, the compiler can add these together <u>at compile time</u>, while generating code in the different subscript macros to add the index factors together. Then, at the very end, it can emit a single instruction to add in the sum A+B+C, or alternatively use this sum as a

displacement in a machine instruction. This is an example of code-generation being deferred over a number of macros. Within the structure, the SS macros keep their results in an 'internally plotted' form, affixed to the macro line. This 'plotted' location would indicate the register in which the index was being computed, and the compile-time number which was keeping track of the sum of fixed displacements.

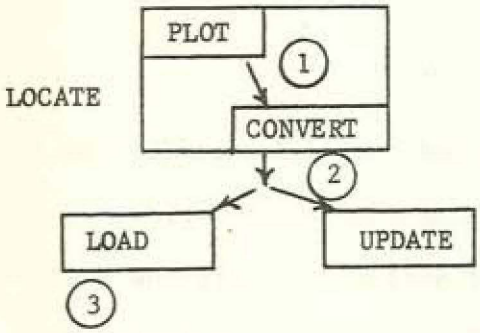


Example 2. A(I).B(J).C(K) where this refers to a structure of packed bytes, where displacements and indices refer to the number of bytes. Array sizes are fixed. Here the SS macro would keep a) a base address, b) a compile time count of displacement in bytes, and c) a computed number of bytes from the base. Before this 'internally plotted location' can be used, (b) will have to be added to (c), and then the result will have to be divided by 4 (bytes/word), and added to (a) while the remainder is kept as a pointer into that address. Clearly, it would be grossly inefficient to do this in every SS macro, since a running total of (b) and (c) can be kept and the conversion from 'internal plotted' form to 'location' need only be done at the very end of the structure reference.

This gives rise to the idea of using a two-part locate function in generating code for structures.

1. The 'PLOT' function which keeps an internal plotted form of location.
2. a 'CONVERT' function which converts this internal form into a 'location'

that can be used by a load/update pair.



LOCATE() = CONVERT(PLOT(PLOT(etc.)))
<value> = LOAD(LOCATE())
<store> = UPDATE(LOCATE())

In an actual implementation, it would probably make sense to have the terminal structure element recognize from the structure table that it was in fact terminal, and apply the convert function itself -- rather than further defer this until the result was used.

1.2.6 THE RELEVANCE OF PLOT, LOCATE, and LOAD:

It might seem at first glance that this division into PLOT, CONVERT, and LOAD/UPDATE is fairly arbitrary, and that there might be other equally valid ways of partitioning the problem. It can be demonstrated that this partition is at least meaningful by considering optimization of common subexpressions.

1. Q := A(I).B(J) .C(K) + A(I).B(J) .D(L)

Here the internally plotted form would give the most efficient optimization.

2. A(I).B(J) := A(I).B(J) + 1

Here the result of the convert (i.e., locate) could be used twice: once with update and once with load.

3. Q := A(I).B(J) + Z (A(I).B(J))

Here the result of the load could be used twice.

In all of these cases, this is true no matter what data-types the various items are.

In fact, you can turn this illustration around, and define PLOT as producing what you need to optimize (1) most efficiently, and define LOCATE and LOAD similarly for (2) and (3).

1.2.7 SUMMARY:

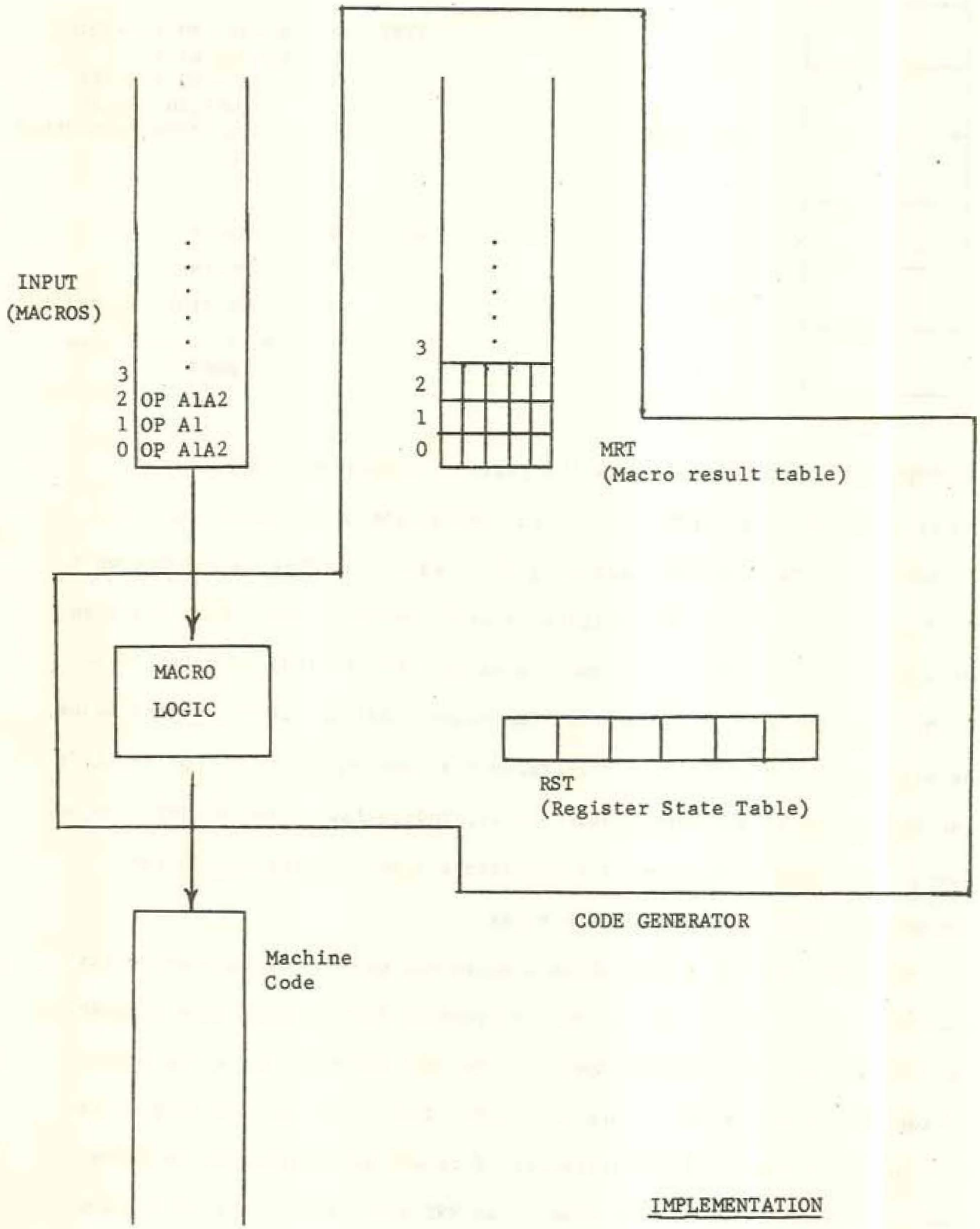
This discussion of data structure macros has outlined the desirability of being able to defer code-generation by letting a macro put out a variety of values to which a code-generating function will later be applied. The next section will demonstrate how this deferal of code-generation can be implemented.

1.3 THE STATE OF THE CODE-GENERATOR (IMPLEMENTATION):

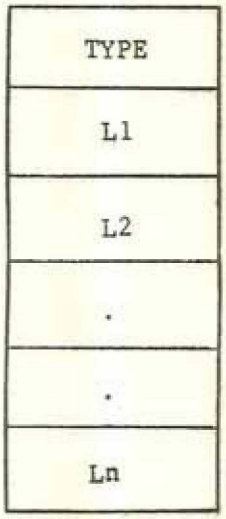
This section describes how the code-generator keeps track of where the various values it has computed are stored at any one time.

The code-generator records its variable state information in two tables. (See figure on following page). The RST (register state table) records what values are in which registers, and also contains additional fields for temporarily locking values into registers. The MRT (macro result table) contains one set of entries for each macro line of input. In it, the code generator records where the various results computed by a macro are stored (i.e., which registers or temporaries), and also records compile-time information for data-structure macros. The exact format of these is described below.

MRT: For each macro line, the MRT contains a set of entries of the following form, describing that macro's results.



IMPLEMENTATION



MRT ENTRY

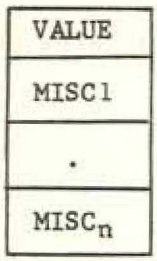
TYPE = s - simply addressable
r - in register
f_i - function required to obtain result
p_i - an internally plotted form
Λ - empty

Li = r_j - register j
t_i - temporary i
l_k - some other location
n - some compile-time parameter

If TYPE is r, then L1 indicates the register containing the value: L2-Ln are empty. If TYPE is s, L1-L3 contain the base, index and displacement of the location containing that value. If TYPE is a function f, then the rest of the entry contains ordered arguments which that function will use. If argument i is a runtime value, then Li indicates its location. If argument i is a compile-time value, then Li itself is that value. As discussed in section 1.2, f represents a load/update pair, and the Li's can be considered arguments passed to it, indicating a complex location in core. When TYPE is P_i, the Li's represents some 'internally plotted' format used within structure references.

The MRT contains fields which a macro can use to indicate where its computed result is, and to put out information for deferred code generation. In practice this information need only be kept recorded for a few intervening macros before the values are used, after which the MRT entries can be considered empty. If optimization of common subexpressions is being done, of course, a value specified by an MRT entry might have to be used several times before that entry could be considered empty.

RST: For each register, the RST contains a set of entries of the following form:



VALUE - a pointer to the MRT entry (matrix line no., entry no.)
- Λ (empty)

$MISC_i$ - miscellaneous information for locking values into registers

The RST is used for recording what registers are currently in use, and what values they contain. This information is used by the GETREG function to locate a free register. When this function has to store a register, it uses the pointer (the VALUE) to update the correct MRT entry so that the entry then contains the temporary into which the value was stored.

The RST is accessed by the GETREG function via a class-definition table which indicates which registers are in the various classes.

USES OF THESE TABLES: The code generator uses these tables as outlined below:

1. Computation macros - When a macro outputs code to perform some computation, it sets any registers containing operands to Λ (unless that value is to be reused), it sets any MRT entries for the operands to Λ (unless it is to be reused, in which case it merely decrements a count), it updates the appropriate entry in its MRT cell to the result of the computation, and it puts a pointer to this entry into the appropriate place in the RST.
2. When the GETREG function stores a register into a temporary, it automatically sets the appropriate MRT entry to contain that temporary.

In this way looking at the MRT will tell you exactly where every value which has been computed but not yet fully used is located.

1.4 SUMMARY:

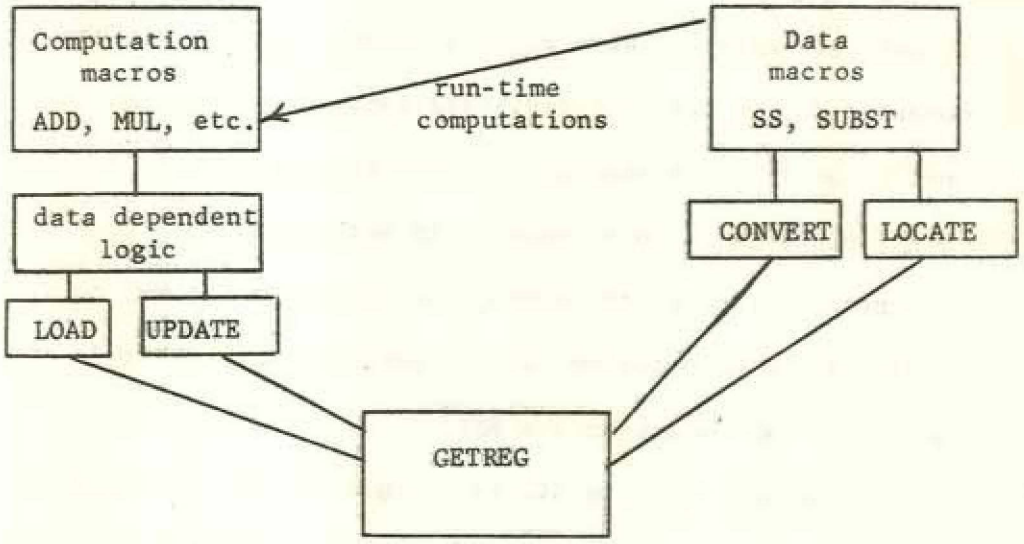
In summary, a code generator consists of the following routines and tables:

ROUTINES:

1. User source macros: including
   a. computation macros (such as ADD) which can be written independently of data-dependent information
   b. data structure macros (such as SUBSCRIPT) which fit into the PLOT-LOCATE framework
2. Code-generating utility routines: to handle the data-dependent logic.
3. A GETREG function: to obtain free registers of a given class
4. LOAD/UPDATE and CONVERT functions: which can be applied to 'locations'.

TABLES:

1. MRT: which records where every relevant computed value is at any time
2. RST: which records the state of the registers.



Rough Hierarchy of Routines

CHAPTER 2:

2.1 A CODE GENERATOR AS A STATE MACHINE:

This chapter describes a code generator as a state machine whose state is determined by the location of the values which are to be used in generating code. These values may be in simply addressable core location, in non-simply addressable core locations, or in a register, of which there may be several classes.

Each computational macro has associated with it certain permitted initial states for its operands. For the 360, for instance, initial states for an integer ADD macro would be: 1. both operands in registers, or 2. one in a register and the other in core. The process of generating code for such a macro, therefore, is the process of the code generator making a trasition into one of these initial states, followed by the emission of a particular code sequence from that state.

Computation macros are thus described in terms of permitted initial 'states', a code sequence to be emitted from each state, and a location where the result of the computation will be left.

There are several advantages to this more descriptive sort of macro system. It makes the job of writing macros simpler since it eliminates the repetitive testing (as described in section 1.1) in each macro to ascertain the state of the operands, since this is done by the system. Also, by making code-generating a more descriptive, rather than procedural, process, it is a step towards automatic code generation, in which the code-generator need only be fed some suitable description of the object machine. This system provides a good environment within which to explore some of the problems involved in automatic code generation for a broad class of object machines.

2.1.2 AN OVERVIEW OF THE STATE MACHINE:

This proposed 'state machine' model rests on three conceptual levels of looking at code generation.

1. The code generator is seen as a state machine. Its operation is seen as one of making successive transitions into a sequence of initial states required for emission of code for a given sequence of operations. This picture of code generation is independent of whether the user specifies these transitions procedurally, or whether the system deduces them automatically from descriptive information.
2. For a class of object machines, it is possible to create an automatic mechanism to perform these transitions from any arbitrary state into a permitted initial state. The fact that such an automatic method can be specified, irrespective of efficiency, is significant in itself, since it points the way to more automatic code generation, and provides a framework for evaluating cost and efficiency problems of such a system in detail.
3. This mechanism can have rules built into it for choosing which of several permitted initial states to aim for, and which of several pathways to use to get there, based on cost information and efficiency considerations. The degree to which these rules are successful determines how efficient the code will be. In a practical system, this might be the most critical aspect. In fact, there does not seem to be any inherent reason that this sort of system should not be able to produce efficient code.

2.1.3 USING THE STATE-MACHINE:

There would be two stages in writing a macro in DMACS taking advantage of this state machine concept. First, the language writer would use MIML to create a machine independent macro. Then a machine user would fill out this macro with OMML declarations. As an example, consider an ADD macro. If there was only fixed arithmetic in the language, the MIML code would be trivial:

```
macro   ADD  X,Y      (commutative)
        ADD  X,Y
```

This indicates that part of an OMML program labeled ADD is to be used in expanding this macro. For the 360 this piece of OMML code would be:

```
ADD A1,A2:
from  REG(A1),REG(A2)  emit AR A1,A2  result REG(A1)
from  REG(A1),WORD(A2) emit A  A1,A2  result REG(A1)
```

Notice that this OMML code is a non-procedural description of 360 integer arithmetic. Together with this OMML section, the machine user would have also given a description of his machines register structure and of paths between it and core.

If the ADD macro was to handle floating arithmetic as well, the MIML program would have to be more complex. It would check to see whether the operands were fixed or floating and indicate different OMML sections for the two cases: i.e., ADD X,Y and FADD X,Y. The macro might also handle conversion between the two modes.

2.1.4 TYPICAL MACROS:

Typical macros might be defined as follows for integer arithmetic on an IBM-360-like machine with a complement-register instruction.

MIML code:

```
macro     ADD X,Y     (commutative)*
          ADD X,Y

macro     SUB X,Y
          SUB X,Y

macro     MUL X,Y     (commutative)
          MUL X,Y
```

OMML code:

```
          ADD A1,A2:
from      REG*(A1),REG(A2)     emit  AR  A1,A2    result  REG(A1)
from      REG(A1),WORD*(A2)    emit  A   A1,A2    result  REG(A1)
          SUB S1,S2:
from      REG(S1),REG(S2)      emit  SR  S1,S2    result  REG(S1)
from      REG(S1),WORD(S2)     emit  S   S1,S2    result  REG(S1)
from      REG(S2),WORD(S1)     emit  COMPL S2;A S2,S1 result REG(S2)
          MUL M1,M2:
from      ODDREG*(M1),REG(M2)  emit  MR  EPAIR(M1),M2 result REG(M1)
from      ODDREG(M1),WORD(M2)  emit  M   EPAIR(M1),M2 result REG(M1)
```

Notice that the programmer only specifies permitted initial states for code emission. For each macro call, the code-generator has to check if one of these states exists, and if not, it must have the necessary machinery to attain one of these permitted states, by generating the appropriate code.

---

* (REG and ODDREG are user defined register classes. EPAIR is a register-register relation. WORD is a user defined memory class. 'Commutative' tells the system that the computation is symmetric, and therefore to also accept REG(A2),WORD(A1), for instance, in the case of the ADD macro.)

2.2.1 IMPLEMENTING THE STATE MACHINE:

This chapter will outline how DMACS is organized to implement the state machine concept. This organization has several parts.

1. A register-structure description: in which the machine user defines his registers, classes over these registers, and data paths among registers and between registers and core.
2. A GETREG function which is deduced from this register-structure description.
3. An algorithm for choosing a target permitted initial state to aim for, for each possible input state, in the automatic transition.
4. An algorithm for sequencing the application of transformations to accomplish this transition.
5. A general algorithm for locking and half-locking values into registers in preparation for code emission from a permitted state.
6. A top-level algorithm to coordinate a macro execution: i.e., the automatic transition followed by code emission.

The remainder of this chapter will discuss each of these topics in turn.

2.2.2 REGISTERS:

Registers are the system resources which the code generator must manipulate to attain permitted initial states. Therefore the user must be able to describe registers flexibly enough to include a large class of object machines, yet with enough restrictions so that GETREG logic can be generated from this description automatically. In particular the system must be able:

1. To obtain a free register of a given type.
2. To store any register.

3. To load any register.

4. To transfer any value from a register where it might be left by a computational macro into any other register where it then might be required.

To allow the system to do this, the user defines a register structure, and indicates the data-pathways, both within it, and connecting it to simple memory (for temporaries). There are a few simplifying assumptions made as to what this structure looks like, but these assumptions follow intuitive notions of object machine structure. The user defines:

1. Registers r_i $i = i,m$

2. Classes over these registers R_i $i = 1,m$

   st. $\forall r_i(\exists R_j \text{ st. } r_i \in R_j)$

   and $\forall R_k, \forall R_{j\neq i}(R_i \cap R_j = R_i \text{ or } R_j \text{ or nil})$

In other words, every register is in at least one class, if only by itself. Also any two classes are either subsets or disjoint. There is no partial overlap.

3. S_{init}: The subset of these classes which might hold computed results is called R_{res}. Therefore, the initial arbitrary input state of the macros is taken from $S_{init}=R_{res} \cup MEM$, where MEM is core memory.

4. Pathways to core: Each class of registers is assumed to have a direct path to and from core. There is no need to go through a second register in either loading or storing. The user must specify what these pathways are.

5. Paths from R_{res} to other registers: The user must specify possible register to register moves from, and within, R_{res}.

6. Relationships between registers: The user can define specific relationships between registers (such as even-odd pairs). He can also specify that when using one under a given class name, the other must be stored and made available as well.

Thus the user describes a register structure, and defines paths within this structure and between it and core. Using this information, the system will construct a GETREG function for the various register classes, and will have the machinery to restore any change that that function makes to the register structure's state.

2.2.3 THE GETREG FUNCTION:

The function GETREG(REGTYPE) returns a free register of the type 'REGTYPE', or alternately a free register-pair, for instance. This routine has the ability to store values out of registers (and update the MRT). When any function is called which in turn calls GETREG, it is not necessarily known what state the code generator will be left in, since values may have been stored.

The logic of the GETREG function is quite straight-forward. It consists of cycling through a given class of registers attempting to find an empty one. If there are none, it must decide which register to store based on the 'flags' attached to the various registers. These flags are used to lock certain values into registers in direct preparation for outputing code, and to half-lock values so that they will not be stored during a given macro expansion unless necessary. The actual mechanism for flagging requires some thought, since macros can be called as subroutines during a given macro expansion. Flags set by the different macros in a single source macro expansion would have to be separable. (An algorithm to perform this is given in section 2.2.7).

Another problem that the GETREG function has to handle is a situation like an even-odd register pair, when two registers must be freed simultaneously. To be able to deduce this sort of logic from a machine description, it is necessary to try to anticipate this sort of relationship between registers.

2.2.4 SAMPLE REGISTER DESCRIPTION: IBM-360

```
rclass REG:r2,r3,r4,r5,r6,r7,r8,r9,r10,r11

rclass ODDREG:r3,r5,r7,r9,r11

relation EPAIR (stored:ODDREG)

     r3:r2
     r5:r4
     r7:r6
     r9:r8
     r11:r10

rpath WORD→REG:     L REG,WORD
rpath REG→WORD:     ST REG,WORD
rpath REG→ODDREG:   LR ODDREG,REG
```

This defines two register classes. For each member of ODD, a related EPAIR register is defined, which is to be stored when the ODDREG member is used as ODDREG. Paths between storage and registers are described.

2.2.5 INPUT TO THE CODE GENERATOR:

Input to the code generator is a series of macros of the form: OP A1 .... An, where OP is the macro name and Ai is the ith argument. Arguments can be described by a triple, (t,f,p), where:

t (type) is s - the argument is simply addressable
r_i - the argument is in register r_i
R_if - the argument will be in a register of class R_i after applying the function f.

f (function) is a code-generating function (transformation), possibly nil, required to generate the actual value represented by the by the operand

p (pointer) is a pointer, either to the symbol table or to results left in the MRT by a previous macro. The information pointed to will be used by f.

The code generator will use this information to automatically generate a permitted initial state for a given macro.

2.2.6 THE AUTOMATIC TRANSFORM:

There are two steps in the process of automatic transformation. First - choosing a target state to aim for, and second - sequencing the application of functions to the macro's operands to attain that state.

I. Choosing Target States: Choosing target states is quite straightforward. Each operand of a macro represents a value which is either in core, in a register, or pointed to by a complex address to which a LOAD function can be applied.

The target selection algorithm is designed for macros of two operands, but the ideas could be expanded to handle more operands easily. This algorithm maps each two-operand pair (from the set of possible inputs) into a permitted initial state. If there is only one such permissible state, then the selection process is already done.

The most general way to do this is, for a given input state, to minimize, over the permitted states, the sum of

1. the execution times of the instructions of the macro, plus
2. the execution time for the pathway to that initial state. (i.e., the loads, stores, or register-to-register transfers).

The algorithm given here merely minimizes the number of instructions in the pathway.

For a given input state, the algorithm uses the function 'Compare' to obtain the cost (in number of instructions) of transforming that input to

each permitted state. The target for that state is set to be the permissible state with lowest cost.

The Compare function determines where the two states differ, and how many instructions it would take to readjust operands.

Examples: input	permitted state	cost in instructions
r,s	R,s	0
s,s	R,s	1 load
r,s	R',s	1* load register
R_f,s	R',s	0*
R_f,R_f	R,s	1 store
r,s	s,r	2 store, load
etc.		

(In this example, r is a register in class R but not R' and R' is a subset of R, s refers to an operand in simply addressable memory. The cost function is based on a 360-like computer.)

*The only minor idiosyncracy of this algorithm is that if one component of the input pair is of the form 'R_f' (indicating that the operand is complex but can be generated into a register of type R), and its final state is a register of type R', where R' is a subset of R, then it is assumed that the value will be generated directly into an R' register by the function f, and therefore the cost (due to this operand) is 0.

The selection of a target state for each initial state pair need not be done every time the macro is executed. It can be compiled into a table when the macro definition is processed.

Algorithm to select target state:

I = set of possible input states, $i \in (S \cup Rj \cup Rjf) X (S \cup Rj \cup Rjf)$
where S = MEM and Rj $\in$ SINIT

P = set of permitted initial states $p \in (S \cup Rj) X (S \cup Rj)$
where s μ MEM and Rj is any register class
(P is assumed ordered for convenience)

Problem: to map each i into one p

1. if $|P| = 1$ then $\forall$ $i \in I$ target (i) = $p \in P$

2. for each $i \in I$ do:
   a. save1 $\leftarrow P_1$
      save2 $\leftarrow$ compare (i, p_1)
   b. for $Pj \in P$ $j>1$ do:
      1. temp $\leftarrow$ compare (i, pj)
      2. temp: save2 if greater or equal do nothing
         if less save1 $\leftarrow$ Pj
         save2 $\leftarrow$ temp
   c. target (i) = save1

finished

Compare (i,p)

1. cost $\leftarrow$ 0
2. (each argument is a two-tuple)
   for n = 1 and 2 do:
   a. if i_n = Pn do nothing
   b. if (i_n is of form Rif and Pn is of form Rj and Rj is a subset of R_1) do nothing
   c. cost $\leftarrow$ cost = Instr (i_n, Pn)

finished

Instr (I2, I2) determines from a table the number of instructions in the pathway between the storage classes.

II. Sequencing the functions: There are two different strategies for sequencing the functions. One is 'blind sequencing' which is used when enough is not known about the functions to predict what they will do. The other is 'controlled sequencing' which is used when information is available about how the functions will use registers, and can be used to sequence them 'more optimally'.

1. Blind sequencing: This is the general case which could handle an arbitrary register structure where the effects of GETLOC were not easily predictable. In this case, it would be possible for the code generator to use certain optimizing rules at macro definition time to actually compile a graph -- which for each input state would tell which operand to apply what functions to and in what order.

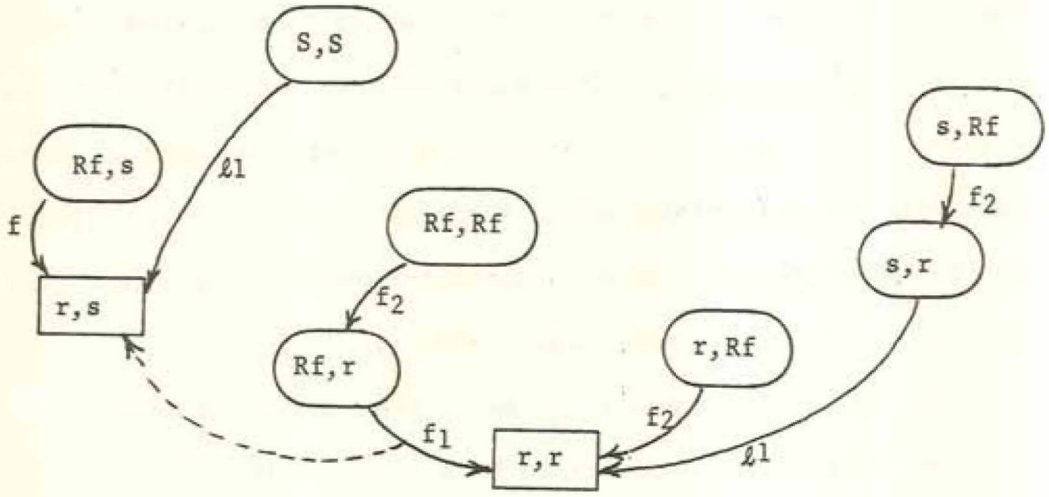
The graph has a node for each combination of $(s \cup R_i \cup R_i f)^2$. For each node, the blind sequencing algorithm decides which function (or load of store) to apply first, and then draws an arc to the node representing the result of that application. Since each arc goes to another node in the graph, this sequencing algorithm need only determine for each state the first function to apply. These arcs are labelled by a tuple indicating what the transformation is and which operand is involved. (see figure) In the examples given, the permitted states are enclosed in rectangles, and the application of functions (either f, or load, or store) are implicit in the labelling of the arcs. The dotted arcs are explained in the following paragraph.

One problem in constructing this sort of graph is that if an operand value is in a register when a function is applied to another operand, then the first operand may be stored.

BLIND SEQUENCING GRAPHS

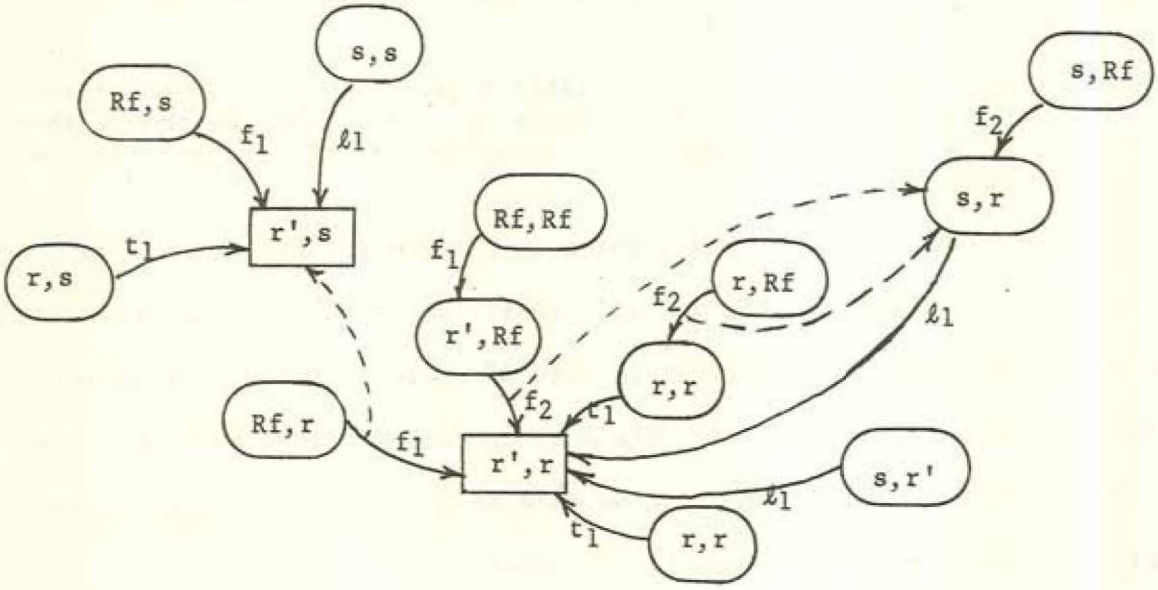
I. INPUT: (S ∪ r ∪ R f) X (s ∪ r ∪ R f)

FINAL: (r,s)(r,r)



II. INPUT: (s ∪ r ∪ r' ∪ R f) X (s ∪ r ∪ r' ∪ R f) r'∈R' R'⊂R

FINAL: (r',s),(r',r)



When this happens, an accidental transition has occured. As a result, in the general case, this graph must be augmented to include transitions, in such 'unstable' situations, to a state where the 'unstable' value is in core. These transitions are represented on the graph by dotted arcs.

Algorithm for blind sequencing: This algorithm for sequencing the graph is quite ad-hoc. It consists of examining each state together with the the target state and determining which function to apply first. This algorithm is applied to each state to determine what arc to draw from it. Applying this algorithm to every state completes the graph.

I. 1. If any operand is r and is to be s, then apply store to it.

2. If any operand is Rf and is to be s, then apply f to it.

3. If only one operand is to be in r, and it is s or Rf, then apply load or f respectively.

4. If only one operand requires a function applies, apply it.

5. (Both operands require complex function and are to be in registers). If there is an alternate permitted state, differing from target only in that one value is in core, then apply f to that operand first. (Then, if the 'unstable' value is stored, you are still in a permitted state.)

6. If one operand requires a register class that is a subset of the class required by the other, apply it first, (on the theory that there will be more chance of finding one of the more restrictive class first).

II. For all above, if there was a value in a register when a function was applied, put a dotted 'accidental' transition to the appropriate state. (The only additional consideration is that if a simple variable requires a base or index register loaded, this must be loaded before emitting code).

This blind sequencing need not be done at every macro-execution, but merely when the macro definition is processed.

Controlled Sequencing: If it is known how many registers are needed by the functions, and how many are required by the results, and a method is available for flagging registers so that they would not be stored, it would

often be possible to determine a sequence in which the functions could be applied, with results locked into registers, so that no deadlocks would occur. (A deadlock might occur if a number of values were irrevocably locked into registers in such a way that the register needs of a later function were unsatisfiable.)

For situations where even-odd pairs of registers were used by the function, this determination might not be quite as clean and linear. In such cases there might be a 'blind' indeterminate range where it was unclear whether a given sequence would work. But even in such a case, attempting controlled sequencing would give some degree of optimization.

Algorithm: The general outline of the algorithm is:

AR = available registers
res(i) = number of registers used by results of i^{th} operand
use(i) = number of registers used by generation of i^{th} operand
F_1 = res(1) + use (2)
F_2 = res(2) + use(1)

1. if AR > F_1 and AR>> F_2 then use 'blind' sequence
2. else if $F_1 < F_2$ apply f_1 first else apply f_2 first

This controlled sequencing clearly must be done independently for each macro execution.

2.2.7 GENERAL PROBLEM OF TRANSFORM:

Automatic transformation looks at the type of its operands, looks at the permissible target states, and then initiates one or more transformations to achieve one of these states. This process can be described in general terms. It is the problem of:

1. having an environment containing certain resources (registers) in a given state (containing certain values).
2. wanting to transform these resources into a new state with certain properties (certain values in registers).

3. having local functions which utilize the resources while transforming them, thus altering the resources in a possibly unpredictable way, while effecting a desired local change to them.
4. desiring to make a sequence of such local transformations and still have the resulting global state well-deined (i.e., either the originally desired final state, or a permissible alternative).
5. allowing these local transformations to communicate (via the Register State Table -- locking values into registers, etc.)

To accomplish this goal, the mechanism that generates initial states must be able to detect when one function it applies stores a register that it expected to be loaded, and either reload that value, or else pick a different target initial state. There are two potential problems in a general system of this sort: 1. static deadlock and 2. thrashing (dynamic deadlock).

In the case of the code generator, this blind, unpredictable mode of operation evantually degenerates to a simpler situation where values are simply addressable and where the results of various operations are well defined since simply addressable temporaries are used. Assuming that enough registers do exist to handle the final results, controlled sequencing can eventually be done with the necessary results securely locked into registers. In other words, blind sequencing, even if necessary, eventually degenerates into controlled sequencing.

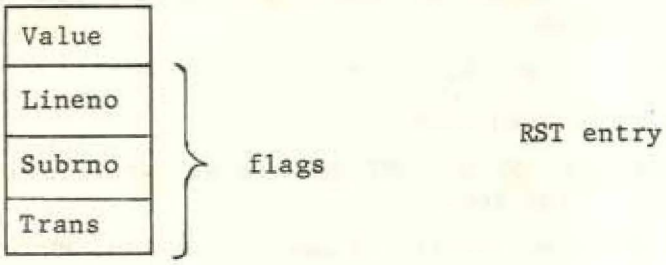
The role of communication between these local functions via the RST is an important one. In blind sequencing, the best that can be done is to half-lock critical values into registers, so that these will be stored only in last resort. In controlled sequencing, when you know that enough

resources exist for the functions you apply, this half-lock is effectively a full-lock and values can be counted on to stay in registers. This implies that an integral part of expanding a macro is half-locking any appropriate values into the RST, before applying any of the functions.

Algorithm to prepare a macro for emission:

Associated with each macro is the current source macro line number: CURLINE. Since several macro subroutines can generate co e during the expansion of one source macro, a global number (MACROSUBR) is incremented for each macro call. Since a macro subroutine call could be recursive, a local variable CURSUBR is provided to hold MACROSUBR for each macro.

This algorithm locks values into registers by flagging the RST entries:



The algorithm also uses a global vector called ACCIDENT, set initially to zero, which is used to record when accidental transitions occur. Entries in this vector are set by the GETREG function.

```
MACRO:  OP  A1,A2

1.   MACROSUBR ← MACROSUBR + 1
     CURSUBR ←   MACROSUBR

2.   for n = A1,A2 do:

     a.  is n to be in a register?
         yes - is n in that register?
               lineno ← CURLINE
               subrno ← CURSUBR
               trans  ← 1
```

b. is n indexed or based?
yes - is index or base in correct register
yes - in the RST entry for index or base
lineno ← CURLINE
subno ← CURSUBR

3. (sequence the application of functions):

a. if controlled sequencing can be done, then do it

b. if the blind sequencing graph is to be used

1. go to first node

2. if node is a final node go to 4.a

3. apply function

4. if an accidental transition occurred
(i.e., if ACCIDENT (CURSUBR) = 1) then follow dotted arc
else follow normal arc to next node

5. go to 3.b.2

4. (all values are in correct locations):

a. load any necessary index and base registers for simply addressable operands

b. erase all locks set by this macro in 2 above

c. output correct code sequence

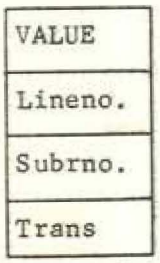
d. erase operands from RST and MRT (unless subexpression optimization is being done).

e. place record of macro result, if any, in RST and MRT

Algorithm of the GETREG function:

GETREG (REGTYPE)

This function cycles through the 'REGTYPE' class of registers in the RST performing tests. Each RST element has the following form:



Value is a pointer to the MRT or is empty Λ

CURLINE is the current source macro line number.

CURSUBR is the number of the macro which has issued the GETREG.

Algorithm for single register: look at each class member for:

1. an empty one - if so return

else 2. one with Lineno ≠ CURLINE

- if so store it in temporary, update MRT and return

else 3. one with Subrno ≠ Cursubr

- if so store it in temporary, update MRT and return

else 4. store anyone, update MRT return

(if trans = 1 for any register stored, then set ACCIDENT [Subrno] = 1)

Algorithm for register pair: this algorithm is very similar

- first look for a free pair
- then for a pair with at worst a different lineno. (if possible half empty)
- then for a pair with at worst a different Subrno. (if possible half empty, else if possible half different lineno.)
- else any pair

(if trans = 1 for any register stored then set ACCIDENT [Subrno] = 1)

2.3 OPERATIONS TO MEMORY:

A useful extension to the state machine concept, as outlined, would be to allow operations-to-memory. These are a common class of instructions and it would be straightforward to incorporate them. The user would be allowed to specify alternate destinations for a value, i.e., for the PDP 10:

```
ADD X,Y
    from  REG (X), REG (Y)      emit  ADD X,Y     result  REG (X)
    from  REG (X), MEM (Y)      emit  ADD X,Y     result  REG (X)
                  OR            emit  ADDM X,Y    result  MEM (Y)
```

Only four alternations need be made to the logic outlined in this chapter.

1. In outputting code for a macro: when one of these alternatives occurs and the MEM is a temporary, defer the code generation and flag the RST entry indicating the two operations and the MEM location.
2. In the GETREG logic: when looking for a register to store -- generate any OPM instruction in preference to storing some value explicitly.
3. In the target selection algorithm: when an input value is deferred, evaluate both possible input states and choose the one with lowest cost. If the lowest cost option is 'OP-to-register' continue to defer generating the code until it is clear that the value need not be stored.
4. As preparation for a macro, generate any necessary OP-to-register code of this sort for operands which have been deferred.

CHAPTER III: DESCRIPTIVE DATA REFERENCE MACROS

3.1 THE NATURE OF DATA REFERENCE MACROS:

This chapter represents a first attempt to develop a framework which would let programming language data-references be automatically implemented for an arbitrary machine. The class of machine structures which this chapter deals with is not very broad, but hopefully could be extended.

The automatic state transition discussed in Chapter 2 allows a machine user to generate computation macros from descriptive information. The problem of acheiving the same kind of machine independence for data structure macros must be solved by a different approach.

When writing data structure macros for a particular machine, one has the following general flowchart:

```
Branch on data-type into:

    Simple:  ---
    Byte:  ---
    Bit:  ---
    (etc)
```

Here, the user knows exactly what data-types he will be dealing with and knows exactly how each is mapped into core: ie. the machine addressability, how the data-item fits into Hardware-addressable units, how to generate its 'location', and how to move it between core and registers. All of this information in contained implicitly in the coding of the macro.

On the other hand, although the exact formats are different, and the factors needed to convert indices to actual core locations are different, the conceptual operations which are performed are very similar for the various data-types. Thus it would be possible for the user to write machine-independent data macros by specifying in the macro definition the conceptual operations which are performed on the operands, and elsewhere specify in descriptive form the machine-dependent information, which

will be needed to flesh out these operations for the various data-types, such as:

1. What the different classes of storage are (bits, bytes, words,double-words, etc.), how big each is, what boundaries they start on, and how they map into one another.
2. What the basic addressable item is (byte?, word?).
3. The instructions or instruction sequences needed to transfer the different data-types between core and registers (ie. load, store, insert byte, deposit byte, masking, shifting, etc.).
4. How the different user data-types map into this memory hierarchy.

The specification of these machine dependent details would be straight-forward since it is done descriptively. The crucial advantage of this separation of macro and machine-description, however, is that it allows the macro to be written independently of the machine, and then be expanded by a description of the machine memery and of the data-types used.

3.2 COMPILER ORGANIZATION:

This chapter describes how a compiler can be organized to yield machine independence for complex data references. It does not describe the internal workings of the DMACS system. Rather it describes a framework which the language writer could use in organizing his compiler to attain this independence. Then the chapter shows how this framework could be incorporated into DMACS to handle data-structures with fixed bounds. With such structures, all the core allocation could be done at compile-time. and therefore machine code need not be generated to compute at run-time data-item offsets within structure elements.

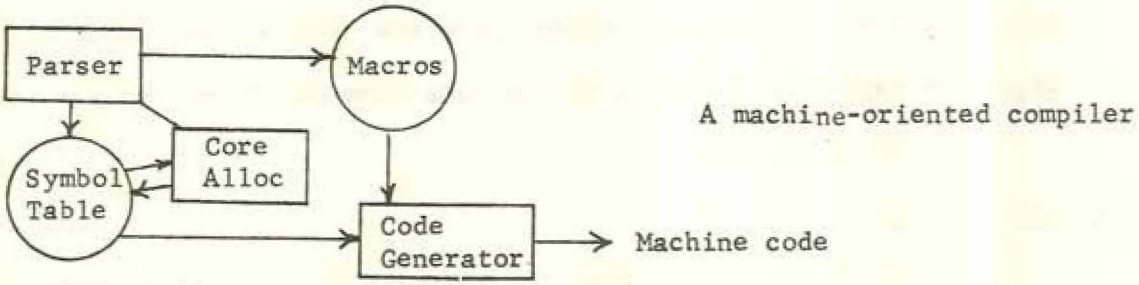
The framework to be discussed involves:

1. Certain assumptions about machine addrrssing structure.
2. A general algorithm which would take machine-descriptive information

and compute, for a data structure, offsets and lengths for each data-item, and determine what data pathway is to be used to access that item, and other auxilliary information.

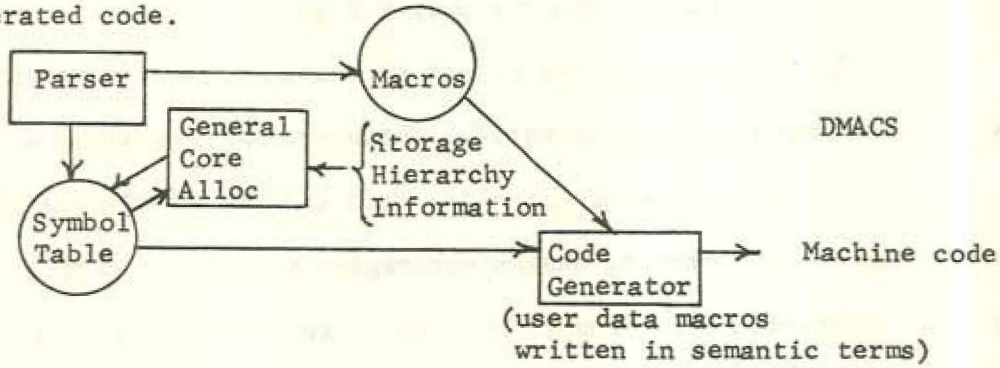
3. Data reference macros which are set up to use the information computed in 2 to determine how to handle a given data-item on a particular machine. A particular data-item might be handled by different sections of logic on different machines. (This is the key concept). Thus a macro can be pictured as being written using 'semantic data-types', and the algorithm in 2. can be seen as mapping actual data-types into semantic data-types for a given machine, and computing semantic primitives describing these.

This approach differs from a machine-oriented compiler in several ways.



A machine-oriented compiler

In a machine-oriented compiler, the core allocator is generally built into the parser, and its operation is seen as being quite separated from the code generating logic, although the results it computes are part of the generated code.



DMACS

In DMACS, the core allocator is, conceptually at least, divorced from the Parser. It operates on the symbol table after being fed descriptive information about a machine. Information which it puts onto the symbol

table (semantic primitive information) will be used directly by the user data-macro logic when generating code.

3.3 ASSUMPTIONS ABOUT THE STORAGE HIERARCHY:

Storage-classes: there exist a number of storage classes: c_i

Storage hierarchy: these classes are hierarchically ordered; as in bits, bytes, words, double-words

lowest class: there exists a lowest class- called 'bits', and all higher classes contain an integral number of bits

addressable unit: There exists one member of this hierarchy with the attribute 'addressable unit'

path: for each of these classes there are paths between core and registers

data-item mapping: a user language data-item can be mapped into this storage hierarchy

Assumptions:

1. all data-items smaller than the addressable unit are accessed by the bitstring pathway
2. Some data-items larger than the addressable unit can be accessed by the bitstring pathway on some machines (ie. 2-byte items on the 360)
3. no data-item is larger than the target register

These assumptions about the nature of machine memory addressing and accessing are far from comprehensive. There are many machines which could not be fit into this framework at all. This is especially true of small machines where bit-conserving design strategies sometimes result in unusual addressing schemes. The assumptions made in this chapter are oriented towards

a fixed word machine of the IBM 360, PDP 10, GE 645 variety. It would be interesting to explore the problems of expanding this data-macro discussion to include some more anomolous memory structures, but that will not be done in this paper.

3.4 SEMANTIC DATA-TYPES:

The assumptions that we have made imply that indexing and offsetting is either to be done in addressable units or in bits. As a result, when manipulating indices, a macro can assume that the lengths of data-items are expressed either in addressable units or in bits.

Therefore data-items can be considered as being of two different 'semantic' types in these manipulations.

1. single-unit items: items which were either one addressable unit long or one bit long.

2. multiple-unit items: items whose length is either a multiple of addresaable units or a multiple of bits. Associated with such an item is a number N_m which represents the length of the item in single units. Thus, in a subscript macro, for instance, the index must be multiplied by N_m.

Using this partitioning of data-items, a subscript macro could generate a proper index pointing into a data base. There still remains the problem of how to normalize this pointer so that it can be used to access the data-item. Consider the problem of accessing a bit-string on the GE 645 and the IBM 360. Assume that you have the address of the base of the data area and a bit index into it. (If the base of the data area includes a bit displacement, then this must be added to the bit index). On the 645, you want to divide that index by 36 (the word length), use the quotient as a full-word index, and use the remainder as the bit displacement. On the 360, you want to obtain

the address of a full-word boundary, plus a bit displacement. To do this, you could divide the index by 32 (the word length), and the remainder would be a bit displacement. Then multiplying the quotient by 4 would yield an index in addressable units. (This assumes that the base of the area was full word alligned).

This discussion implies that a data item also may have the following attributes:

N_d- a number to divide a bit pointer by, to yield a bit pointer as a remainder

N_a- a number to multiply the result of that division by to yield addressable units

N_r- a number to multiply the remainder by to yield an expression in bits

(N_a and N_r may be 1);

These numbers N_m, N_d, N_a, N_r may be considered to be semantic primitives. A given data item would be characterized differently by these on different machines, and hence would be handles by a different section of the data macro logic.

The following example shows how such a semantic macro could be coded for subscripting with fixed array sizes.

Sample macro logic: (Subscript)

```
SS   X,I

is size of X fixed?
     yes-- is X a single unit item
          yes-- put into MRT locate(X,I)
          no-- multiply I by Nm yielding In
               put into MRT locate(X,In)
     finished
```

where locate(X,I) is

```
is I in bits (to be used by the bitstring pathway)?
     no-- put into MRT: Load/Update 'type', X,I
     yes-- divide E by Nd yielding Id as quotient, Ir as remainder
          is Na=1
```

yes-- put into MRT: L/U type, X, Id, Ir, length
no -- multiply Id by N_a yielding Ia; put out L/U,X,Ia,Ir,length

Notice that a given data-item might be handled by different parts of this logic for different machines. For instance, on the IBM 360 the byte is the addresaable unit and would be handled by the single unit logic, where-as on the word-addressable GE 645, a byte item would be handled as a bit string by the logic for multiple unit items.

3.5 STORAGE MAPPING:

Part of the user's description of the memory hierarchy of his machine involves describing how his data-types are to be mapped into this hierarchy. Some data-types may start automatically on a particular boundary. Other data-types may start automatically on some boundary when the user has specified the attribute 'alligned'.

Since the machine user specifies this information to the code generator, and can specify different sets of this information (different machine descriptions), obviously no storage allocation of any sort can be built into the parser. Only after processing a machine description, can it be computed just how big different data-items are, and where they are located, what semantic data-types they are, and what semantic attributes (N_m etc.) they have.

An algorithm is presented in section 3.7 to process a data-structure using a machine description, and produce this information for the elements. The algorithm outlined is designed to perform these computations for arrays with fixed bounds. It can be thought of as a preprocessing of the structure declarations, which 'primes the pump' for the machine-independent data macros.

3.6 COMPILE-TIME COMPUTATIONS:

The previous discussion has presupposed that array sizes are known at compile time. In languages like PL1, this need not be so. The problem that arises when array sizes are not known is that all displacements below that array cannot be known until runtime. Hence, in accessing any of these, code must be ganerated to calculate displacements for these elements.

The exact nature of these computations and depends on the implementation of the language. The problem of making these references machine independent is different from the fixed array problem, since the algorithm described in 3.7 must be implemented partly in a procedural macro language, and partly in a higher level language which could be compiled into machine code for a particular machine. This problem is a sticky one, but still retains the basic concepts discussed for acheiving machine independence in data reference macros.

3.7 MACHINE INDEPENDENT CORE ALLOCATION ALGORITHM:

The following algorithm is a rough outline and may need some refinement. Its purpose is to accept first, a machine description, and second a set of data and structure declarations, and then to allocate core positions where possible, and also to compute the semantic primitives for that data-machine pair. These primitives include: lengths of items in either addressable units or in bits, semantic type (single or units), N_m (if appropriate), and the appropriate load/update pathway to access that item on that machine.

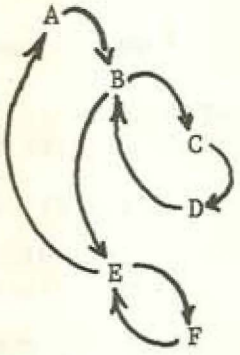
This algorithm makes several assumptions. Array sizes are fixed. If a structure element itself is not aligned, then nothing within it is to be aligned. Each structure element, if aligned on an addressable boundary will be given a length in addressable units, otherwise in bits.

The algorithm expects that a data-element will be described by information (constructed by the parser) indicating: its data-type, whether it is aligned, whether it is a terminal element (leaf) of a structure or not, its length, whether it is indexed, and if so what the indexing bounds are.

The algorithm works on a structure recursively, determining absolute addresses if possible for data-items. (For data items within an indexed structure element, this is not possible). In doing this recursive analysis, the algorithm determines the length of <u>each</u> sub-element on a given level and then backs up to the next highest level.

Structure

```
1 A
 2 B
    3 C
    3 D
 2E
    3 F
```



In performing this analysis the algorithm uses a 3 element vector to record displacements and lengths. This has the following format:

DISP:

ADDR is the number of addressable units

BITS is the number of bits

Flag can be 1. 'absolute' meaning that it indicates actual core location
2. boundary X indicating that the displacement is relative but starting at boundary X
3. 'bits' meaning only bit lengths are being counted, and no alignment is being done.

Algorithm

1. set DISP vector to address of beginning of data area, absolute

2. call ALLOC (DISP) which returns LENGTH
   add LENGTH to DISP

3. is this last data-item on this level
   -yes- stop
   -no - go to 2.

ALLOC (disp) Local: L-DISP, NEXT, TYPE, MEMTYPE, LEN

1. L-DISP ← disp

2. NEXT ← next item declared

   TYPE ← type of NEXT (from parser)

   MEMTYPE ← storage class that TYPE maps into
   (from machine description)

3. is NEXT to be aligned? (from parser input)

   - yes - is L-DISP at correct boundary?
   - no - increase L-DISP to correct boundary

4. call Analyze (L-DISP, NEXT) which returns LEN
   add LEN to L-DISP

5. is this last element at this level
   no - go to 2.
   yes - return L-DISP

Analyze (disp, next) LOCAL:DISP,LEN

I. is next a structure element?

yes - 1. is it aligned?

no - set DISP to 0, 0, bit
yes- set DISP to 0, 0, alignment

2. call ALLOC(DISP) which returns LEN for the sub-element of the structure

3. is this element indexed?

yes - a. if element is aligned and LEN doesn't end on correct boundary then increase LEN to boundary

b. multiply LEN by index number

c. record N_m

4. if next is terminal perform II.1 and II.2

5. return LEN

II. (not a structure element)

1. if disp is absolute then record disp as address of item

2. put out bit length of item, if relevant, and the correct load/update pair for MEMTYPE

3. return length of item

3.7 A ROUGH EXAMPLE STORAGE DESCRIPTION

```
mem BIT
mem BYTE (8 BIT, addressable)
mem WORD (4 BYTE, 32 BIT, boundary 00)

mpath   WORD→REG:                       L     REG,WORD
mpath   REG→WORD:                       ST    REG,WORD
mpath   BYTE→REG:                       SR    REG,REG;IC  REG,BYTE
mpath   REG→BYTE:                       STC   REG,BYTE

mpath   BIT→REG:        (WORD,DISP,LEN)
                                        L     REG,WORD
                                        SLL   REG,DISP
                                        SRL   REG,32-LEN

mpath   BIT→REG:        (WORD,REG(DISP),LEN)
                                        L     REG,WORD
                                        SLL   REG,REG(DISP)
                                        SRL   REG,32-LEN
```

```
mpath    REG←BIT:        (WORD,DISP,LEN)
                                    L     ODDREG,ALLONES
                                    XR    EPAIR(),EPAIR()
                                    SLDL  EPAIR(),DISP
                                    SLL   EPAIR(),LEN
                                    SLDL  EPAIR(),32-DISP-LEN
                                    L     REG,WORD
                                    NR    REG,EPAIR
                                    SHL   R,32-DISP
                                    OR    REG,R
                                    ST    REG,WORD
map BIT to BIT (align:WORD)
map CHAR to BYTE (align:BYTE)
map FIXED to WORD (align:WORD)
```

This storage description is a partial rough outline of what is needed.

Not included are:

1. load/update routines for bit fields which run across word boundaries
2. the inclusion of some kind of runtime computation facility to perform calculations like 32-DISP-LEN when DISP is passed as a runtime value.

Also, in a practical system more flexibility might be desired such as the ability to perform some of the more complex load/update routines as subroutines.

CHAPTER IV: 'PROCEDURAL' VS 'FUNCTIONAL' DESCRIPTION

4.1 'FUNCTIONAL' vs 'SEQUENTIAL' DESCRIPTION:

In chapters 2 and 3 we have looked at how computational macros, data macros, and register manipulation logic can be generated from descriptive information.

A good deal of this information described the structure of the machine (the register structure and the memory hierarchy), and the functions that individual instructions perform within this structure. The description of load/update routines, on the other hand, involved the user specifying an ordered sequence of instructions to perform together a specific function. Describing instruction sequences in not the same as merely describing the machine instructions available.

A distinction can be made, then, between two modes of description. A 'functional' description system would be one in which the user merely described his register and memory structure and instruction set. Such functional description would assume a whole 'machine state'. The user would describe how each machine instruction operates on this state: for instance, what indicators a compare instructions sets, whether it skips, (increments the control count) and what indicators a Branch-on-condition instruction tests. From this sort of description, the system would have to deduce how all the little pieces interrelated, and would have to construct primitive code sequences from this deduction.

A 'sequential' description system, on the other hand, would be one in which the user was required to give some information implicitly by ordering instructions into very low-level sequential primitives. This sequential description system is perhaps less elegant in some unclear sense.

It does not put any real burden on the user, however, and it does make the job of creating such a system much easier.

It can be seen that the task of creating a functional description system for code generation can be separated into two stages. First the creation of a system in which primitives could be given in sequential form. Then once these primitives were isolated, deductive routines could be included to deduce these primitives from functional information. It might be interesting to try to deduce such sequential primitives from a hardware register-transfer-language description of a machine.

Load/update routines, for instance, might be deduced from some functional description of shifting and masking. Similarly, 'compare-and-branch' primitives discussed in the next section might be deduced from some functional description of condition-codes, automatic skipping, comparison, and conditional branching.

DMACS is a 'sequential' system, at least at present. As the 'sequential primitives' needed become more clearly recognized, however, it may be possible to include routines to deduce them from descriptive information.

4.2 SEQUENTIAL PRIMITIVES:

Load/update routines are sequential primitives which system requires to be described. The user in creating machine-independent macros can create his own. A good example of a case where this might be necessary is the testing and branching facilities of the machine.

Consider a 'compare' macro which generates a result of true (1) or false (2). For the 360, this would be accomplished by the following instruction sequence:

```
REG (G1), REG (G2)          CR    G1, G2    REG(1)
                            BG    .+10
                            SR    G1, G1
                            B     .+6
                            LA    G1, 1
```

This is not a satisfactory solution to the problem, however, because it is not a machine-independent macro definition. To allow the code for such a macro to be different for different machines, suitable sequential primitives must be defined. An example of such a macro is

```
macro:    GT  G1, G2 (inverse:  LE)
          COMPJG G1, G2, .+3
          SETO
          BRANCH  .+2
          SET1
```

This macro involves four primitives which will be filled out by the machine user (COMPJG, SETO, BRANCH, SET1). DMACS would perform appropriate mapping into instruction sizes on the object machine, so correct addresses would be branched to.

In describing his machine, the user would then give meaning to these primitives. For the 360 these could be:

```
COMPJG  C1, C2, C3
from    REG (C1), REG (C2)  emit CR  C1, C2
                                 B   C3
from    REG (C1), MEM (C2)  emit C   C1, C2
                                 B   C3

SETO        emit  SR  REG, Reg      result  REG
SET1        emit  LA  REG, 1        result  REG
BRANCH B1         B   B1
```

This approach lets the user describe his machine in a compact and quite descriptive form while allowing the macro to be written in machine independent form.

4.3 SEQUENTIAL PRIMITIVES FOR SUBROUTINE CALLS:

Allowing the user to write machine independent subroutine call macros (CALL, ARG, ENTRY, RTN) does not require any elaborate machinery as did computation or data structure macros. The reason for this is that there is a very direct mapping between the operations which the source macro specifies and the actual machine instructions.

To allow flexibility as to object machine structure, however, care must be taken to assure sufficient machine independence. For instance, the stack should be allocated in the correct increments of addressable units. The nature of the stack pointer should be flexible: either a position in core, a special register, or a set-aside register of a given class.

To help the user attain this flexibility, four system-deduced functions are provided: .MOVE, .MOVEA, .ADDR, and .STACK. .MOVE will move its first operand from core or from a register to the address specified as its second operand. .MOVEA is like .MOVE except it moves the address of its final argument. .ADDR will convert a number (i.e., of words) to the proper number of addressable units. .STACK represents the stack pointer. The expression (.STACK) acts as a base.

Using these primitives, subroutine macros might be defined roughly as follows: (these may not be sufficiently general)

```
MACRO     ARG  A1
        .MOVE A1, (.STACK) ENDFRAME + N
        n ← N + 1

MACRO   CALL  C.
        N ← 0
        .MOVE  .STACK, (.STACK) ENDFRAME - 1
        .MOVEA .+3, (.STACK) ENDFRAME - 2
        .MOVEA (.STACK) ENDFRAME, .STACK
        J      C1
        .MOVE  (.STACK) - 1, .STACK
```

```
MACRO   ENTRY
        STOREREGS

MACRO   RTN  R1
        .MOVE  R1, (.STACK)
        LOADREGS
        JIND (.STACK) - 2
```

Notice that some of the problems we are getting into here are not local code generation problems, but rather general machine-independence problems of a more global nature. It is not really within the scope of this paper to address these. For a given real-world language, the compiler writer might want specific primitives to be defined for linkage pointers, argument pointers, etc. This paper has only gone as far as applying this technique to the stack. Real world languages might require a more broadly designed system.

4.4 THIS SYSTEM VS. UNCOL:

In the real world, there are a large number of programming languages and a large number of object machines. Automatic code generation as described in this paper is an attempt to simplify this situation.

If DMACS is successful, a language written with a machine independent macro code generator would be able to run on a broad class of machines. It is not quite so clear how simple the inverse of this language to machine adaptation would be. In other words, once a machine-user had written a description of his machine for one code-generator, how much work would he have to do to convert this description so that a language with a different descriptive code-generator would accept it? Presumably, the register and memory hierarchy, and computational instructions would be described the same way. Obviously, the mapping of data-types into memory would have to change if the data types were different. To the degree that the second language used different sequential primitives, this part would have to change, but with some standardization, this change could be kept minimal. Hopefully then, any changes would be small ones and rapidly done. On the other hand,

if the language was radically different, the changes might be more substantial, but this is not unreasonable.

It is instructive to compare this approach to the problem of proliferating languages and machines to that embodied UNCOL, an attempt at creating a universal macro-like intermediate language. The hope was that any high-level language could first be compiled into UNCOL and the UNCOL could be compiled into any machine code. Thus, complete program transferability could be ensured at the cost of only one translator per language and one code generator per machine. In practice, the restrictions of having only one universal intermediate language have proved too confusing to be easily solved, at least as yet.

DMACS sidesteps this problem in two ways:

1. It allows the user to specify his own macro language. Thus the macro language can be tailored to the source language.
2. By letting the user deal with sequential primitives it creates a standard interface very close to machine language.

As a result, the process of adjusting to this interface (writing different sequential primitives for different compilers) is logically very simple and straightforward for someone familiar with his machine but not with the language.

Notice that if DMACS were a 'functional description' language, taking as input some register-transfer description of a machine to fill out its sequential primitives, then this description could presumably be input unchanged to any code generator. This would therefore solve the UNCOL problem, by using a machine description as an interface, instead of an intermediate macro language.

CHAPTER V: SUMMARY OF RESULTS AND AREAS FOR FURTHER THOUGHT

5.1 SUMMARY OF RESULTS:

I. This paper develops a machine independent model of the process of code-generation. This model pictures the code generator as:

1. a state machine which makes repeated transitions into permitted states from which it can emit machine instruction.
2. operating on conceptual (semantic) data-types.
3. built on low-level semantic primitives.

This abstract machine-independent model is a step towards a more formal definition of the code generation process.

II. The paper shows how an implementation of this machine-independent model could be set up to generate code for a number of different machines, from machine-dependent information, given in a descriptive form.

III. The paper discusses some aspects of how a compiler (parser and code generator) should be designed to allow this kind of machine-independence.

Thus, the paper is a step towards formalizing code generation, and abstracting it from any particular machine. This is similar to the formalization of parsing which allows automatic creation of parsers from a BNF description of a language.

5.2 FURTHER THOUGHT:

This paper has attempted to set down main ideas and to expand upon a number of them. Some specific problems, therefore, still require some thought.

It would be instructive to look through all of the constructs of a powerful language like PL/I, and see what problems com up in fitting them into the general automatic framework outlined in this paper.

It would also be interesting to consider the problem of deducing semantic primitives from something similar to a hardware register transfer description of a machine's instruction set. This problem would be a substantial piece of research in itself.

5.3 GENERALIZATION OF THIS MODEL:

The model presented in this paper is a restricted one in several ways. It deals only with register machines and it assumes that the machine has index registers. Also, it is not specifically designed to handle operands of the sort required for 360 decimal arithmetic, for instance. It would therefore by worthwhile to extend the model to include a broader class of machine structures. In particular, it would be useful to extend it to stack machines, to machines with no registers, and to machines with more anomalous addressing schemes.

It would also be very relevent to consider how the framework discussed here could interface with more global aspects of optimization, such as common-subexpression optimization, removal of invariance from loops, and optimal register allocation over loops.

APPENDIX I:

This section contains a set of macros which handle computation and control, written in MIML (Machine-Independent Macro Language). It also contains part of two machine descriptions written in OMML (Object Machine Macro Language), which flesh out these macros.

To write an entire code generator, data macros and subroutine macros would have to be added to the MIML code and register and storage descriptions and additional primitives would have to be added to the OMML declarations.

SAMPLE MACHINE INDEPENDENT MACRO DEFINITIONS:

Note: the integer in parentheses after a macro name indicates that macro's number. i.e., the uniform symbol indicating an ADD macro would have an index of 1.

```
MACRO ADD(1) X,Y (COMMUTATIVE)
             ADD

MACRO SUB(2) X,Y
             SUB

MACRO ASSGN(3) X,Y
       .UPDATE (X,Y)

MACRO MPY(4) X,Y (COMMUTATIVE)
             MPY

MACRO DIV(5) X,Y
             DIV

MACRO GT(6) X,Y (INVERSE: LE)
       CJG X,Y,.+3
       SET0
       J .+2
       SET1

MACRO LT(7) X,Y (INVERSE: GE)
       CJL X,Y,.+3; SET0; J .+2; SET1

MACRO GE(8) X,Y (INVERSE: LT)
       CJGE X,Y,.+3; SET0; J.+2; SET1

MACRO LE(9) X,Y (INVERSE: GT)
       CJLE X,Y,.+3; SET0; J.+2; SET0

MACRO EQ(10) X.Y (COMMUTATIVE)
       CJE X,Y,.+2; SET0; J .+2; SET1

MACRO NE(11) X,Y (COMMUTATIVE)
       CJNE X,Y,.+3; SET0; .+2; SET1

MACRO JMP(12) X
       J X

MACRO TJMP(13) X,Y
       CJE X, =1,Y

MACRO FJMP(14) X,Y
       CJE X, =0,Y
```

PDP 10:

```
ADD X,Y:    from  REG(X),REG(Y)    emit  ADD X,Y        result  REG(X)
            from  REG(X),WORD(Y)   emit  ADD X,Y        result  REG(X)
                                   or    emit  ADDM X,Y       result  WORD(Y)
SUB X,Y     from  REG(X),REG(Y)    emit  SUB X,Y        result  REG(X)
            from  REG(X),WORD(Y)   emit  SUB X,Y        result  REG(X)
                                   or    emit  SUBM X,Y       result  WORD(Y)
            from  REG(Y),WORD(X)   emit  MOVC Y,Y;ADD Y,X  result  REG(Y)
                                   or    emit  MOVC Y,Y;ADDM Y,X result WORD(X)
MPY X,Y:    from  REG(X),REG(Y)    emit  IMUL X,Y       result  REG(X)
            from  REG(X),WORD(Y)   emit  IMUL X,Y       result  REG(X)
                                   or    emit  IMULM X,Y      result  WORD(Y)
DIV X,Y:    from  REG(X),REG(Y)    emit  IDIV X,Y       result  REG(X)
            from  REG(X),WORD(Y)   emit  IDIV X,Y       result  REG(X)
                                   or    emit  IDIVM X,Y      result  WORD(Y)

CJG c1,c2,c3
          CAMLE c1,c2
          JRST c3

CJL c1,c2,c3
          CAMGE c1,c2
          JRST c3

CJGE c1,c2,c3
          CAML c1,c2
          JRST c3

CJLE c1,c2,c3
          CAMG c1,c2
          JRST c3

CJE c1,c2,c3
          CAMN c1,c2
          JRST c3

CJNE c1,c2,c3
          CAME c1,c2
          JRST c3

SET0:       from  nil              emit  MOVEI s1,0     result  REG(s1)

SET1:       from  nil              emit  MOVEI s1,1     result  REG(s1)

J   j1:     JRST   j1
```

IBM 360:

```
ADD:      from  REG(X),REG(Y)    emit  AR X,Y          result  REG(X)
          from  REG(X),WORD(Y)   emit  A  X,Y          result  REG(X)

SUB:      from  REG(X),REG(Y)    emit  SR X,Y          result  REG(X)
          from  REG(X),WORD(Y)   emit  S  X,Y          result  REG(X)

MPY:      from  ODDREG(X),REG(Y)emit  MR EPAIR(X),Y    result  REG(X)
          from ODDREG(X),WORD(Y)emit  M  EPAIR(X),Y    result  REG(X)

DIV:      from  ODDREG(X),REG(Y)emit  DR EPAIR(X),Y    result  REG(X)
          from ODDREG(X),WORD(Y)emit  D  EPAIR(X),Y    result  REG(X)

REM:      from  ODDREG(X),REG(Y)emit  DR EPAIR(X),Y    result  REG(X)
          from ODDREG(X),WORD(Y)emit  D  EPAIR(X),Y    result  EPAIR(X)

CJG       C1,C2,C3
          COMP C1,C2
          BG  C3

CJL       C1,C2,C3
          COMP C1,C2
          BL  C3

CJGE      c1,c2,c3
          COMP c1,c2
          BGE  c3

CJLE      c1,c2,c3
          COMP c1,c2
          BLE  c3

CJNE      c1,c2,c3
          COMP c1,c2
          BNE  c3

CJE       c1,c2,c3
          COMP c1,c2
          BE  c3

COMP      c1,c2:
          from  REG(c1),REG(c2)        emit   CR c1,c2
          from  REG(c1),WORD(c2)       emit   C  c1,c2

SET1:     from    nil                  emit  LI sl,1     result   REG(sl)
                                  or    emit  MVC sl,=F   result   WORD(sl)

SET0:     from    nil                  emit  XR sl,sl    result   REG(sl)
                                  or    emit  MVC sl,=F(0)result   WORD(sl)

J jl:   J   jl
```

APPENDIX II:

This section contains a rough first description of the MIML and OMML BNF.

MIML: Machine Independent Macro Language

⟨PROG⟩ ::= ⟨DECLARATION⟩ * ⟨MACRO⟩ * end

A program is a series of declarations followed by a series of macros.

⟨DECLARATION⟩ ::= dcl ⟨DCL⟩

⟨DCL⟩ ::= externaltable (NUM) NUM ⟨.IDN⟩*

| manifest (NUM) IDN ⟨.IDN⟩*

The user declares any external tables he might want to reference, tells what type it is, how many fields it has, and optionally gives names by which he can refer to the fields. He also declares any manifest constants, giving type numbers for them. Variables are automatically allocated and need not be declared.

⟨MACRO⟩ ::= macro IDN (NUM) IDN ⟨,IDN⟩*{(⟨ATT⟩⟨;⟨ATT⟩⟩*)}$^{-1}$⟨MDEF⟩

⟨ATT⟩ ::= commutative | inverse : IDN

A macro consists of a macro name followed by its number in the macro definition table, followed by any arguments, followed by any attributes, and finally followed by the macro definition.

⟨MDEF⟩ ::= IDN | .IDN {IDN ⟨,IDN⟩*}$^{-1}$

|⟨PRIMITIVE⟩ ⟨;⟨PRIMITIVE⟩⟩*|⟨LOGIC⟩

⟨PRIMITIVE⟩ ::= IDN {⟨ARG⟩ ⟨,⟨ARG⟩⟩*}$^{-1}$

⟨ARG⟩ ::= IDN | . {+ NUM | - NUM}$^{-1}$ | = NUM

A macro definition can specify that OMML text be inserted, that a system function (.IDN) be applied, that a primitive be expanded, or that logic be executed.

⟨LOGIC⟩ ::= ⟨ST⟩ | [⟨ST⟩ $^{+}$]

⟨ST⟩ ::= if ⟨BOOL⟩ do ⟨LOGIC⟩ | ⟨COMMAND⟩

⟨BOOL⟩ ::= IDN | ⟨VAL⟩ $\overset{\leq}{>}$ ⟨VAL⟩ | .IDN {(IDN ⟨,IDN⟩*)}$^{-1}$

⟨VAL⟩ ::= ⟨VAR⟩ | NUM | ⟨VAL⟩ $\overset{-}{+}_{/*}$ ⟨VAL⟩

⟨VAR⟩ ::= {mrt | IDN} (IDN) . IDN | IDN

⟨COMMAND⟩ ::= .IDN (IDN ⟨,IDN⟩ *) | IDN (IDN ⟨,IDN⟩ *)

|break|⟨VAR⟩ ← ⟨VAL⟩

OMML: OBJECT MACHINE MACRO LANGUAGE

⟨PROG⟩ ::= ⟨DECL⟩*⟨REGDECL⟩ ⟨MEMDECL⟩ ⟨INSTDEF⟩*⟨PRIMDEF⟩*END

⟨DECL⟩ ::= manifest (NUM) (LENGTH:NUM) IDN ⟨,IDN⟩*

⟨REGDECL⟩ ::= ⟨CLASS DECL⟩*⟨REGREL⟩*⟨PATHDECL⟩*

⟨CLASSDECL⟩ ::= rclass IDN : IDN ⟨,IDN⟩*

⟨REGREL⟩ ::= rrelation IDN {(⟨ATT⟩ ⟨;⟨ATT⟩⟩*)}$^{-1}$⟨IDN:IDN⟩*

⟨ATT⟩ ::= stored: IDN

⟨PATHDECL⟩ ::= rpath IDN → IDN : IDN IDN ⟨,IDN⟩*

⟨MEMDECL⟩ ::= ⟨STOREDEC⟩*⟨M-PATH⟩*⟨DATAMAP⟩*

⟨STOREDEC⟩ ::= mem IDN {NUM IDN ⟨,NUM IDN⟩*⟨,⟨M-ATT⟩⟩*}

⟨M-ATT⟩ ::= addressable | boundary 0^{+} | picture = {NUM IDN}$^{+}$

⟨M-PATH⟩ ::= m-path IDN ({frommem | tomem} : ⟨INSTR⟩*

⟨DATAMAP⟩ ::= map IDN (NUM) to IDN {(allign : IDN)}$^{-1}$

⟨INSTDEF⟩ ::= IDN : ⟨STATEDEF⟩$^{+}$

⟨ STATEDEF⟩ ::= from IDN (IDN), IDN(IDN) emit ⟨INST⟩* result IDN(IDN)

⟨PRIMDEF⟩ ::= IDN IDN ⟨,IDN⟩*⟨PRIMBODY⟩*

⟨PRIMBODY⟩ ::= ⟨PRIMDEF⟩ | ⟨INSTR⟩ | ⟨STATEDEF⟩*

BIBLIOGRAPHY

(1) Cheatham, T., "Course Notes on Compiling," Applied Math 295, Harvard University.

(2) Strachy, C., "Fundamental Concepts in Programming Languages," Programming Research Group, Oxford University, England.

(3) Graham, R.M., Programming Systems (to be published)

(4) Graham, Malek, Miller, P., Murphy, Sussman, "LPS--A Language Processing System", ProgLing Memo no. 1, Massachusetts Institute of Technology.

(5) Horwitz, Karp, Miller, R., Winograd, "Index Register Allocation," JACM, January 1966.

(6) Orgass, R.J., and Waite, W.H., "A Base for a Mobile Programming System", Comm. ACM, September 1969.

(7) Waite, W.H., "A Language Independent Macro Processor", Comm ACM, July 1967.

(8) Strong, J., et al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution", Comm ACM, August 1958.

(9) Steel, T.B., Jr., "A First Version of UNCOL", Proceedings WJCC 1961, pp. 371-378.