TR80-008

THE PORTABLE DPL COMPILER PROJECT

by

John E. Bishop

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the department of Computer Science.

Chapel Hill

1980

Approved by:

David L. Parnas, Advisor

Peter Calingaert, Reader

Stephen F. Weiss, Reader

THE PORTABLE DPL COMPILER PROJECT

by John E. Bishop (under the direction of David L. Parnas).

Portability of a compiler is achieved by generating code for a pseudo-machine. This code is then translated into code for various real machines. The technique is combined with the use of templates, which are fixed translations of source code structures, to aid in the design of the compiler. A formal use of these templates partially specifies the source-to-object translation. Problems associated with the implementation of these techniques and with the project are documented, and solutions are suggested.

ACKNOWLEDGEMENTS

I would like to acknowledge my debt to Peter Parker (alias Spiderman) for his demonstration that an academic career can come to an end, take it ever so long, and to Howard the Duck, for his example of courage and perseverance in the face of a strange and demanding world. Special thanks are due to my co-workers on the project, particularly Dan and Jim, who introduced me to the mountains.

CONTENTS

Chapter I

INTRODUCTION

1.1 THE DPL COMPILER

In 1976 Edsger W. Dijkstra published A Discipline of Programming [Dijkstra, 1976], in which he introduced and used a procedural algorithmic notation. In this notation, non-determinacy was a property of the major control structures. The fact that the order of evaluation was not specified enabled algorithms to be derived from a problem statement more easily. The notation was not a programming language; the book was not a manual. Despite the lack of formal presentation -- Dijkstra did not even give a full BNF syntax -- the notation was not hard to understand.

Dr. David L. Parnas, of the Computer Science Department of the University of North Carolina at Chapel Hill, initiated a project to develop a compiler that would allow the book's notation to be used as a programming language. We called this language Dijkstra's Programming Language (DPL). Our implementation was to be portable, modular, student-oriented, and provide for inclusion of a program-holder module. Initially Dr. Parnas, Dr. Robert Wagner of Duke University, and two graduate students -- James D. George and I -- were

involved. We started planning the compiler in January of 1978, and by that summer added two other graduate students to aid in more detailed planning and in the writing of code. The next year was spent programming, and by the end of the summer of 1979 the compiler was running. During part of this time we were under the direction of Dr. Mehdi Jazayeri.

1.2 CONTENT OF THIS THESIS

In this thesis I plan to discuss the organization of the compiler, covering the design of the compiler, the method used to achieve machine independence and the effect this method had on the module that generated code (the CG). I also describe a planning tool for code generation called 'templates', and show how it made the task easier but made the code that was generated slightly less efficient. The faults of the compiler as a whole are discussed, and recommendations are suggested.

The module I wrote is described in broad terms in chapter five, and in detail in the last appendix. Readers who are not interested in the internal details of this module need not read either.

There are four appendices. The first is an attribute grammar describing the translation the CG performed. Part of this description is a BNF grammar of the input to the CG. This input, a tokenized form of the source, was the

Intermediate Language (IL). The translation of a construct is given in the form of the template for that construct. The second appendix is a listing of the real values of the elements of the IL, rather than the mnemonics used in the first appendix. The third is a short explanation of the semantics of the Abstract Machine (the AM), for which the code was generated, and of the AM assembly language in which the templates are written. The last appendix is the documentation for the module I wrote.

1.3 POINT OF VIEW

My responsibility was to design and write the module (the CG) that generated all but the array operation code from the intermediate language (IL) produced by the Front End (FE). The code I generated was not machine code for any machine, but a series of calls to another module, the Abstract Machine Assembler (AMA), whose role in achieving independence of a particular machine is detailed later. The tasks of these modules are sketched in the next chapter.

1.4 WHAT THE READER SHOULD KNOW, WHAT WILL BE LEARNED

I assume that any reader knows what a compiler does, and how. Some familiarity with formal grammars is needed, as is an acquaintance with FORTRAN and PL/I. It is not necessary to have read A Discipline of Programming, nor any of the other theses that came out of this project.

The fifth chapter is not part of the discussion of the project. Only those interested in the CG module need read it.

This thesis is about one particular project, and has two goals: to describe the DPL compiler project in particular, and to show how and where it failed or succeeded. I hope that readers may become better software project leaders (or followers) by analyzing this example.

Chapter II

THE DPL PROJECT

2.1 GOALS OF THE PROJECT

Our major goal was the implementation of DPL by a compiler that would allow us to run all of Dijkstra's algorithms as written with the addition of input and output statements to confirm their actions. Dr. Parnas wished to use DPL as the programming language in an introductory programming course at UNC, and thus wished to have a working compiler by the fall of 1978. It became clear that other goals would have to be sacrificed if this deadline were to be met, and delivery was postponed a year.

Correct execution of Dijkstra's algorithms was only one goal of several. A set of goals that was mutually-reinforcing consisted of portability, construction from modules that were independently designed and written, and the use of information-hiding. Other goals were that the compiler would be student-oriented, and allow addition of a program-holder module. We were also interested in the compiler's provability. These goals were not offically ranked, but portability was the most important after the execution of Dijkstra's algorithms, with information-hiding and modularity close

behind. These goals were all met, to some degree. This will be dicussed in chapter five. Efficiency and small size were never goals.

2.1.1 Portability

A portable program is one that is easily made to work on many computers. An example is a ANSI-standard FORTRAN subroutine that calculates cube roots, while an example of a non-portable program is a machine-language keyboard driver usable with only one kind of keyboard with one kind of interface to one particular machine. Since different locations have different machines, it is more complicated for a compiler to be portable: a portable compiler must generate code for these different machines, as well as run on them. This requires either generation of a high-level language or some re-writing of the code-generation module(s). If the compiler is written in a language the local machine cannot use, then the compiler itself must be translated. Our approach tried to alleviate both of these problems. Our compiler was written in ANSI-standard FORTRAN, for which almost every installation has a compiler, and it did not generate code for any particular machine, but for a simple machine we had designed, called the Abstract Machine (AM). This code was designed to be easily translated into the appropriate "real" machine code. This had certain repercussions on the CG, and these will be discussed in chapter five. This

approach to portability is treated in more detail in chapter three.

2.1.2 Modularity

Modularity is a familiar goal [Parnas, 1972]. A large task is divided into sub-tasks called modules. These sub-tasks can be further sub-divided, and those sub-divided, until a sub-task is sufficiently simple that a small program can do it. "Good" divisions are ones that lead to elegant, simple and independent sub-tasks; "bad", those that do not. A module is not necessarily a sub-routine, but subroutines are often part of a module. A good example would be the division of an interpreter into a driver, a program-holder module, a data-storage module, a parser and an operations module.

2.1.3 Information-Hiding

Information-hiding is Dr. Parnas's term for a method of designing modules [Parnas, 1972 (again)]. Each module conceals something. It may be the algorithm, the kind of data structure, or the representation of externally presented information. Such secrets belong to a particular module: no other module may know about them. This means that a user of a module cannot assume a certain internal structure or form of representation, nor can a used module assume that it is used for a specific purpose or by a specific user. The

module and its user share knowledge only of the function the module is to perform. Inter-module communication is thus confined to the explicit interface of calls and parameter passing, or reference to specified global variables. By not specifying the internal workings of a module, we have made it possible to change these workings, without forcing changes to other modules. As the specification of the interface describes all that need be known about the module, and all that its writer need know, the module may be written in isolation, without the need to consult with other programmers.

An example would be the specification of a symbol table by the calls to enter a symbol with certain attributes, or to retrieve the attributes of a symbol. The user neither knows or cares whether the symbol is inserted in a binary tree, hashed into an array, or linked into an unordered list, nor whether the attributes are kept separately, encoded into a bit vector or passed on to another attribute-saving module. If the user knew, for example, that symbols were stored into an alphabetically ordered binary tree, he might try to present the symbols in such an order that the tree was well-balanced. Or the user might want to use the same encoding of the attributes as the symbol table, and the end result is that the symbol table and its user form one module. The two programmers must talk every day, and neither module can be changed without changing the other. Program-

mers have enough information about their own module already; the technique of information-hiding prevents swamping them with information about others. The thesis of information-hiding is that the efficiency lost by hiding a secret is traded for the ease in writing and maintaining modules that do not know it.

But there is a proviso: the interface must be well specified. For a simple module, this can be easy, though even the simplest can have hidden subtleties, but for a complex module, it can be very hard. Consider the specification of a compiler. The attribute grammar appended to this thesis is not enough, as the translation of source names into memory locations is not specified. To specify that requires knowing the rules of scope and variable-reference in DPL. We used English and tables, and had difficulties in determining which module checks which rules (see 6.1.3.2).

2.1.4 Student Orientation

Undergraduates who are taking an introductory programming course are, by definition, unable to understand the workings of a large program like a compiler. If the compiler is written with neophyte use in mind, it will try to communicate with its users in terms of the code that they wrote. Thus, for example, error messages should give the number of the source line in which the error was found, refer to all variables by their source names, and explain the nature of the

error. Our policy of information-hiding made this harder than it otherwise would have been: the CG often knew the nature of the error, the FE knew the name of the variable. Thus printing an error message that used a variable name required the co-operation of both modules, lacking at times (see 6.1.2 and 6.1.3.2).

2.1.5 Provability

If a program is provably correct, it can be mathematically shown that it generates correct output from valid input. If the compiler were proven correct, this would mean that the final machine language program was always a translation of the DPL source and had the same semantics as the source. To show that one program had been correctly translated would not be enough. To prove the compiler, it would have to be shown that any valid DPL program would be correctly translated.

An attempt at starting to show the correctness of the compiler is showing that the AMA code sequences (the 'templates', which we hope the compiler implements) are semantically the same as the DPL constructs they translate. If they are, and if the compiler can be proven to implement them, then it would be proven correct.

An attempt at the task of proving the templates was undertaken by another graduate student at UNC. Whether or

not the compiler was correct, proving it so would be easier--perhaps possible--if it were well specified and divided into isolated modules with fixed interfaces. Thus our interest in provability reinforced our desire to use the technique of information-hiding.

2.1.6 Program Holder

A program holder [Parnas and Robinson, 1973] would hold some form of a DPL (or other language) program, perhaps in a tokenized or parsed form. It would allow editing, preferably interactive, at the source code level. This editing would be syntax-oriented, not character-oriented. Thus one could change all occurrences of the variable 'AA' to 'BB' without changing other occurrences of the character string 'AA'. A program holder is not included as part of the compiler, but the break-down of the task into modules was influenced by our desire to allow the introduction of one later.[1]

2.2 STRUCTURE OF THE COMPILER

2.2.1 Major Division

To achieve these goals the compiler was divided into five modules, each with its own secrets. The Front End (FE) knew the nature of the source language and the names of the vari-

---

[1] A program holder for DPL had been built as one of the projects in the course 'Software Engineering' in the spring of 1978. We knew it could be done.

ables. It also knew the rules for the inheritance of variables from one block into another. The code generator (CG) knew the method of code generation, the inheritance and scope rules, and the nature of the translation (i.e. the templates), while the Array Manager (ARM) knew the method for implementing the DPL extensible arrays. The Abstract Machine Assembler (AMA) knew how the Abstract Machine code was stored, what the real values of the mnemonics used as parameters were, and the timing and order of the production of the Abstract Machine code. Finally, the Abstract Machine (AM) knew how the code was translated into the appropriate actions.[2]

2.2.2 Motivations

This division was motivated, in part, by the nature of the improvements we wished to add later. The division of the FE from the CG at a level close to the source made the Intermediate Language (IL), which was the major interface, more like the source: the disadvantage of this was that the CG had to know a great deal about the rules of DPL, as the FE could not hide these rules by presenting a simpler IL to the CG. If a program-holder were to be added, however, the IL's closeness to the source would become a major asset,

---

[2] This can be accomplished either by interpretation or by further translation to some language (machine code) which the hardware then interprets. For wider coverage of these tactics, see the thesis by James Dalton George [George, 1979].

making the translation from the internal IL to source and back easier.

The division of the Abstract Machine into the AMA and the AM was intended to simplify portablility; at first the functions of the AMA were provided by the AM. The separation of the assembly-like functions is intended to make the re-writing of the AM for different machines an easier task.

We envisioned future changes in the scope rules of the language. Dr. Parnas wished to be able to change these rules without changing the compiler [Parnas, Elliot, and Shore, 1975]. This meant that those parts of the compiler that knew the rules should be limited. To aid in the change, the FE was required to replace all variables in the DPL source by unique symbols in the IL, thus making it possible for the CG to assume that an occurrence of a particular symbol always referred to the same variable, and allowing changes in the scope rules to be restricted in their impact to the FE. The isolation of the ARM made improvements or modifications to the array implementation easier, though we had no specific plans there.

2.3 MODULE GOALS AND SECRETS

2.3.1 The Front End

The FE was required to do the following: provide tokens from the IL version of a program, one at a time, upon request. The IL program was guaranteed to be syntactically correct up to the most recent token. If there were no more source, or if an error were detected in the source, no more tokens would be produced, and a flag would be set to tell the CG that no more IL tokens would be produced. The IL is described in the first appendix by the Backus-Naur grammar elements of the attribute grammar that presents the templates. Its most notable characteristics were that expressions were in postfix notation, and that variables had unique integer names. These names were to be consecutive integers, starting at one.

Another set of functions in the FE was associated with constants in the program: each constant was present in the IL as an integer constant name. Upon inquiry, the length of the constant (number of characters) could be found, and the 'i'th character of any constant could be read. The characters could be read in any order, or re-read.

In short, the goal for the FE[3] was the production of a simpified and unambiguous version of the source, with as many of the syntactic rules of DPL hidden within it as was possible.

---

[3] Further detail on this module is available in the thesis by Karl M. Freund [Freund, 1979].

2.3.2 The Abstract Machine and its Assembler

The AMA provided the CG with a set of callable routines whose parameters looked like the elements of assembly language statements: there were calls to generate code, to reserve and to initialize storage, and to create and define labels. The syntax of these calls was simple and the semantics of the call, as opposed to the semantics of the code generated, were equally so.

An unusual feature was that the name for a label had to be created by a special call before any reference to its name could be used in a code generation call. There could be forward reference to a label once the label had been created. No forward reference to storage was allowed.

The AMA hid the AM completely. The CG did not know the 'real' format or values of AM code, nor when the translation to real machine actions took place. Thus from the point of view of the CG, the AMA _was_ the AM. The goal of the AMA was to provide a flexible and powerful language for the CG, while assuring that the AM would remain simple, and thus ensure portability.

2.3.3 The Array Manager

The Array Manager (ARM) module was large and complex, but its complexity did not affect the rest of the compiler. It consisted of the compile-time routines that generated the

run-time support routines, as well as the compile-time routines that generated the code to link to the support routines. It implemented the array operations of Dijkstra's language, which meant that the ARM hid the secret of the semantics and implementation of these operations within itself. Note that neither the FE nor the CG knew the semantics of array operations, as the FE knew only their syntax, and the CG knew only when the ARM would have to be called.

Like the CG, the ARM generated code through calls to the AMA. These routines were invoked at run-time by calling sequences generated at compile-time by calls to certain functions in the ARM. The interface to the CG might sound complex: a run-time environment with certain parameters in certain registers had to be produced by the code generated at compile-time before the ARM was called at compile-time to generate the run-time calling of an array routine, yet despite the double calling structure, the ARM was easy for the CG to use as its syntax was simple. It was not as easy as it might have been (see 7.1.3.2).

The secrets of the ARM were many and though perhaps interesting in themselves, had no effect on the rest of the project once the planning stage was past. In the beginning, though, it was the knowledge that the ARM would be allocating and copying arrays of AM storage that sparked the creation of special AM operations, of which two are not found in

any machine. One would allocate storage, one free it.* The others copied values from one sub-array to another. As these instructions were not used by the CG, the point of view of this thesis, and as the interior of the ARM was subject to information-hiding, little will be said about the ARM.

2.3.4 The Code Generator

Within the CG there were submodules that hid information from each other, and sub-goals for these modules. The overall goal was the translation of the IL to calls to the AMA, and insuring that the IL program presented was semantically correct. Hidden within the CG was its method, its data structures, and the semantics of DPL. It also isolated the FE from the ARM and the AMA, and the ARM and AMA from the FE.

Given that the FE was to have an interface to the program holder, and the ARM was to implement the infinitely-extensible Dijkstra arrays, and the AMA-AM pair machine independence, one would have expected the CG to be experimental, too. It was designed using information-hiding and modularity, and with student use in mind, and it was written in portable FORTRAN, but the CG was not innovative. Any

* This kind of storage had to be assigned a type (integer, boolean, or character) before it could be used. In this respect, the AM is unlike any real machine (see 3.4.1.2). The assignment of a type was called allocation; un-typing was called freeing.

interesting aspects follow from its place in the compiler, and its relation to the other modules, not from a new parsing or generating method. It did perform alone the traditional task of compilation: it parsed a high-level source language, checked it for errors, and generated code for a machine.

Chapter III

MACHINE INDEPENDENCE

3.1 THE PROBLEM

A cursory definition of machine independence has been given. It is a problem of several inputs and several outputs, which can be called the M-by-N problem: given M input languages, to be translated into N output machine codes, how many translators must be built? In our case, M equals one, Dijkstra's Programming Language, and N equals the number of machines on which will the compiler will eventually run. In general, the straightforward answer is MN translators, one for each possible pairing. There are ways to reduce the total number of translators, one of which gives an M+N solution.

3.2 THE GENERAL INTERMEDIATE LANGUAGE SOLUTION

If we divide each translator into two partial-translators, and add a suitably-defined new language, the first half can be given the task of producing this new language from one of the M inputs, and the second that of translating from the new language to one of the N outputs. The new language is an intermediate between the input and the output languages. This means two translations for each pairing of

of input and output forms where the other had one, but the total number of translators is reduced, to one for each of the M inputs, and one for each of the N outputs. This is the 'intermediate language' solution, and has been widely used.[5]

3.3 OUR INTERMEDIATE LANGUAGES

For M equal to one (our case) there would seem to be no advantage to using an intermediate language as measured in terms of the total number of translators. Our intermediate language, however, is chosen so that writing the N second-half translators is far easier than writing N complete translators.

In fact, we have not one but two intermediate languages: the one that functions as the intermediate for purposes of portability is called Abstract Machine code (AM code), while the other is called Intermediate Language (IL). This latter is a tokenized and processed form of the source that is the input for the CG module. Either or both of these could function as intermediate languages. Thus several input languages could be processed into IL, which could be the input to any number of translation programs, just as is AM code.

---

[5] An early proposal was for UNCOL, a Universal Computer Oriented Language, in 1958 [Strong, et al., 1958], but the idea is older, and may pre-date computers or writing. [George, 1979] lists later references in his bibliography.

This use of IL as an intermediate language is not as likely as is the use of the AM language, because IL is very much like DPL, and not general enough to allow easy translation of an arbitrary high-level language into it. Another language like DPL, or a modification or extension of DPL, could easily be translated into IL, allowing a compiler for such a new language to be built very quickly.

Any intermediate language must be specified both syntactically and semantically. Good methods exist for context-free syntactic definition, and adequate ones for semantic definition (which is usually given the burden of defining the context-sensitive parts of the syntax, too).

3.4 THE AM-CODE AS INTERMEDIATE LANGUAGE

3.4.1 Justification of Features

We designed an intermediate language that looks like an assembly language for a register-file machine (see 3.4.1.1) This decision, and others about the AM, must be justified. Why didn't we choose some other form for AM code? As an example, the IL is already partially in postfix and partially in prefix form, and a little more processing would give a completely pre- or postfixed version of the program, suitable for execution by a stack-oriented interpreter, or translation into real machine code. Quadruples[6] are well

---

[6] Also known as 'three-address-code'. Commands have four

known as an intermediate code, too; why not use them? Again, the AM is defined as having bit, integer, character, pointer and instruction data types, but it could have been defined with floating-point numbers also, or with only strings of bits. Much of the design of the AM may be just the personal preference of the designer, but the foregoing questions show the need for justification of the features of the AM and AMA.[7]

3.4.1.1 Register-File Machine

A 'register-file' machine is one that has two levels of memory: the 'file' is the larger of the two, and allows only limited operations on its contents, while the 'registers' allow a much larger set of operations. Typically, the contents of an element of the file must be read into a register before an arithmetic or logical operation may be performed on it. Some machines sub-divide each of these two levels. The PDP-11/45 and the IBM System/360-75 are both 'register-file' machines.

As an IBM System/360-75 and a PDP-11/45 were available to us in the University at Chapel Hill, we chose a register-file organization for the AM to simplify implementation

---

parts: an operation, the two operand addresses, and an address where the result is to be stored. Thus '+ A,B,C' is the equivalent of the PL/I 'C=A+B;'.

[7] Some answers to these questions may be found in [George, 1979], along with much more detail on the design decisions of the AM.

here. Another reason was the fact that most installations have such a machine. Making AM code close to real machine language makes translation into real machine code easier than for any other form, and generation, rather than interpretation, was our goal. This was one of the few decisions made on the basis of efficiency.[8] The AM had three kinds of memory. Two were the register(s) and file of a register-file machine, and the third was allocated and freed by certain AM operations. This kind of memory was introduced to make the ARM simpler, as well as to allow the AM to check the references to array elements. As I am not familiar with the ARM, I know little of this feature, and will say little of it. It was, however, not efficient.

Thus pre- or postfix code was ruled out, as being unlike real machine language, wheras the use of quadruples was ruled out to make register allocation the responsibility of the CG, not of the programmer of the local AM.

3.4.1.2 Data Types and Strong Typing

Most machines have only one type of storage: a string of bits (usually of specific length: 4, 8, 12, and others). This string can be interpreted in many ways: as a number, characters, boolean values, or (part of) an instruction.

---

[8] James D. George's estimate for his IBM-360 version of the AM was that each AM instruction was translated by 4.2 IBM-360 instructions, while AM data were not expanded in translation. See 6.3.

Which interpretation is being used by a program cannot be told by inspection of the bits. This is not true of the AM memory: each piece of AM storage has a type associated with it, and references to it that try to use it as a different type cause a trap to occur. This association is called "strong typing", and was introduced to enhance the ease with which errors in variable reference could be detected. The AM is a machine that checks operands to ensure they have the correct type before each operation. This helped find errors in the CG. It also simplifies the task of transporting the AM.[9] As bits and integers can not be converted to each other, for example, the number of bits in an integer does not have to be specified. The particular set of types was chosen for convenience; bits, integers and characters occurred in DPL, pointers and instructions in the templates.

3.4.2 Similarity to Machine Language

Though a compiler could generate code with no forward references, it would require the ability to hold at least part of the translated program in the code generator. This function is readily separable from that of code generation, and it was so separated, becoming the task of the AMA (Abstract Machine Assembler). The code generator now can make forward reference to locations in code, which simpli-

---

[9] But the extra expense of run-time checking in the AM makes transporting the compiler more difficult. This part of the AM will probably be ignored (see 6.2).

fies its task, while the AMA resolves them for use by the AM. This means that the AM code looks like an assembly language to the CG, but a machine language to the AM. It should thus be easier to transport.

3.4.3 Differences from Machine Language

The important differences between AM code and real machine code are the fact that a given memory location is bound to a particular type, and that only instructions may have aliases (more than one name). The special 'allocate' and 'free' instructions are another difference. The AM is not a Von Neumann machine as instructions may not be changed by the action of other instructions. Type conversions are usually impossible, as noted in 3.4.1.2. The only conversions that are legal are from the integers 0 through 9 to the corresponding characters and vice versa, and this must be done explicitly with an instruction. This means that tricks available to other compilers like doubling a number by shifting logically left one position are not allowed. The lack of aliasing means that there is only one way to access any memory location that is not an instruction. This helps reduce errors that otherwise could not easily be detected.

These differences enhance portability, as it is precisely in the area of type conversion and instruction-modification that machines differ most. One machine will be like another in the way integers are added (assuming no over- or under-

flow) but they will differ on which bits of the instruction specify the use of an index register, or whether ASCII or EBCDIC is used.

A difference that hinders portability is the special kind of memory allocated and freed by the ARM. The major justification for this feature was that it allowed the AM to know the type of each element at run-time, and thus to check the use of each element. The 'allocate' command gives this kind of memory a type, and 'free' removes the type. Free memory of this kind cannot be accessed. The AM is made larger and more complicated by this feature.

3.5 EFFECTS ON THE REST OF THE COMPILER

The effects on the rest of the compiler were limited by our use of the principles of information-hiding and modularization to one module. This was the user of the AM, the CG. The module in between these two, the AMA, was not affected at all by the design of our machine.

Even so, there were only a few effects on the CG: in the use of forward references as explained previously, and in memory management and the choice of AM instructions the CG produced. There was a minor effect on the register allocation sub-module.

In terms of AM memory management, the strongly typed nature of AM memory names and labels meant that reuse could

not change the type, that arrays could be reused only as arrays of the same size (I speak here of AM arrays, not DPL arrays), and that labels could never be reused. This meant that any program for reusing memory had to segregate all memory by type and arrays by size. For this reason no reuse of memory was assumed in the initial design, though the current version does reuse scalars.[10]

As I did not know the relative costs of instructions or of the various addressing methods, because they changed from one target machine to another, I was not motivated to avoid using an expensive intruction or kind of memory access within an instruction, replacing them by two or more cheaper accesses or instructions. Thus there are no machine-language tricks in the CG,[11] such as using repeated addition rather than multiplication.

Finally, the AM definition did not specify the number of registers, beyond granting at least three for use by the CG. A simple register-allocation algorithm was used to allocate these registers, with an AM parameter to the CG giving the number of registers hidden within the sub-module.

---

[10] The FE detects the potential for reuse; the CG implements it.

[11] Except for the routine CONVRT, which translates IL characters, coming from FORTRAN through the FE, into AM characters, which are AM-defined. This routine should be thought of as part of the AM, rather than the CG, as it is machine-dependent.

Chapter IV

TEMPLATES

4.1 DEFINITION

In machine-shop or sewing terms, a template is a pattern that aids in the cutting or shaping of material.[12] It is usually not of the same material as the final product and it may be different in other ways. For example, a template may be a mirror image, or a negative, or convex where the final product is concave. A simple instance of a template is a straight-edge. The final product is a straight line, on a surface, in pencil, ink or as a groove, while the template is a solid object.

The computer language equivalent to a straight-edge could be a simple macro skeleton: it defines a final product, it is not of the same material (format or language) as the final product, and it can be used again and again. Macro skeletons can become more sophisticated, defining and redefining themselves, with internal variables and conditions, but when they do so, they become a new string-processing language rather than a simple, rigid, unchanging form on

---

[12] Webster's New World Dictionary calls it 'a pattern, usually a thin plate, for forming an accurate copy of an object or shape'.

which a final product can be shaped. They are no longer templates.

4.2 DPL TEMPLATES

Our templates were very much like simple macros. The skeleton or form of the AM translation of an IL source construct was described by giving source constructs with formal parameters, and listing for each construct a sequence of AMA calls, using those parameters, and AM variables created within the macro. Templates included the Abstract Machine Assembler (AMA) calls that generated code (given here as pseudo-assembly statements to avoid explaining the syntax of the AMA), and calls to the Front End (FE) and the Array Manager (ARM), as well as a few simple control statements such as '*for each* object[i] *do* action[i]'.

A simple example of a DPL-style template should clarify the foregoing, and make the the second example, the real template for the 'do od' construct, easier to understand.

Suppose that DPL had a data type of 'COUNTER', with three operations, 'zero', 'increment' and 'decrement'. Then each operation would require a template.[13]

---

[13] The meaning of the AMA language used in these templates is described in detail in appendix C.

```
source:    INCREMENT SYMBOL
template:  LOADF Ra, AM name of SYMBOL
           ADDF  Ra, 1, NR, M
           STOF  Ra, AM name of SYMBOL

source:    DECREMENT SYMBOL
template:  LOADF Ra, AM name of SYMBOL
           SUBF  Ra, 1, NR, M
           STOF  Ra, AM name of SYMBOL

source:    ZERO SYMBOL
template:  LOADF Ra, 0, NR, M
           STOF  Ra, AM name of SYMBOL
```

First, note that while the translation of a construct implies more than generation, such actions as symbol table management and checking of the correctness of variable reference are not included in the templates, as they are implementation-dependent. Also, the templates describe the translation of correct input only.

Further, as register allocation is machine-dependent, it is not explicit either here or in the first appendix. Register management is implied by use of Ra, Rb, or R1, R2, R3 to specify the names of registers. The directive that a certain quantity is to appear in a certain register, or to appear in a register that is not a certain register, is seen in the template for the 'do od' construct below. However, in the templates we used during planning and coding, register allocation and freeing were explicit, as we had decided

on a particular allocation method (described in section V.1.7).

```
source:     DOOD one or more (n) of
                [ <expression i> <lstmtln i> ] ENDLST

template:          JUMPL  START, NR, D
   for each of the n sets <expression i> <lstmtln i>
    [ GUARDi: STOP   R1, SAVE, NR, D
              template for <expression i>
                    into register Ra
              CJCB   Ra, TRUE, NR, D, NE, SAVE, NR, I
              template for <lstmtln i>
              JUMPL  SUCCESS, NR, D
        ]

  GUARDS <--  GUARD1, GUARD2, ..., GUARDn
  GNUM        DC     n
  SAVE        DS     POI
  SUCCESS:
  START:      MMMP   GUARDS, NR, M, SNAME, NR, D
              MMMF   GNUM, NR, M, SNUM, NR, D
              JLKL   SHUFL, NR, D
              LOADF  Rb, [n - 1], NR, M
  LOOP:       JLKL   R1, GUARDS, Ri, XI
              SUBF   Rb, 1, NR, M
              CJCF   Rb, 0, NR, M, GE, LOOP, NR, D

                    Rb may not be R1
```

Here a template refers to other templates, and there are local variables, such as 'SAVE' and 'GNUM', as well as parameters such as 'n', the number of guards, and the use of control statements.

4.3 USES OF TEMPLATES

These templates are only a way for describing translations, and a tool for thinking about these translations of

source language constructs, and not an original invention.[14] Many compiler-writers may already use some similar method of specifying the translation. However, we tried to use templates in a formal, consistent way, writing templates for each of the constructs, trying to find errors in the templates, trying to simplify them, and make relationships in the source language, as between the 'if fi' and 'do od' constructs, imply relationships in the templates and thus in the AM code. Only when this had been done did I start coding the CG. We used templates as planning tools and a record of the translation we intended to make, and as a formal specification that could be proven correct. For if a template could be proven formally to be semantically equivalent to the source construct, then that part of the compiler would be proven correct, insofar as it realized the translation specified by the template. The attempt to prove the templates uncovered several errors that might have otherwise gone undetected until late in system-testing time.[15] This shows the worth of templates in the debugging process.

---

[14] Templates have various names in different books. Calingaert uses the macro analogy, and calls them 'skeletons' [Calingaert, p183]. Aho and Ullman, treating the more general case of 'syntax directed translation', call them 'semantic actions', and give a template-like example for simple expressions [Aho and Ullman, pp245-295, with the example pp 266-267].

[15] These were sometimes very subtle errors which would have been very hard to discover, let alone track down. However, the compiler that was proven correct is the one that existed at a certain time: later changes, though minor, may have invalidated this proof. See Steve Bellovin's dissertation [Bellovin, in preparation].

4.4 ADVANTAGES

It was as a planning and design tool that the templates were most useful. They could be developed and changed independently. We could postpone definition of the exact translations of sub-constructs while defining the translation of major constructs. Templates were also independent of the method chosen to parse the IL or of the method chosen to generate code. As templates specified only the result, they were independent of the structure of the compiler, and yet could be used to predict problems and to determine what information would have to be available at a given point in the compilation. A template could say, for example, that a certain construct required two distinct registers, or a jump to the code that was the translation of the construct that followed this one, or that a certain construct would require its own save location. If there was difficulty in writing a template, this might mean that the source construct being considered for translation was poorly divided into subconstructs, and that we should try rewriting the source grammar. Last, the AM code for a given construct could be investigated for possible optimizations, for functions that could become sub-templates, and, in the early stages of design, for often used code sequences that might be candidates for a new AM instruction.

4.5 DISADVANTAGES

The templates we used, despite their virtues, did not take the place of designing and writing the actual compiler, as they do not specify a program, a representation of data, or the various support routines that a compiler must have (such as a symbol table, a register allocator, or memory manager). The actions the compiler is to take may be specified, but how these actions are to be accomplished is not.

But the real objection aims at the heart of any use of templates: they are too rigid and too simplistic, and make no allowances for inter-construct optimization. Not only do they not allow for the detection of common sub-expressions, they do not even allow for the simple avoidance of jumps-to-jumps, or elimination of store-load pairs. Indeed, if one investigates the templates in the first appendix, he will notice that all arithmetic operands are loaded into registers, whereas many loads could be saved by not loading the right-hand operand, and using another of the memory addressing options. This is an inefficiency we avoided in the CG, but not by modifying the templates (but this could have been done, as is explained in the next section).

4.6 FIX-UPS FOR TEMPLATES

A simple answer to these objections might follow the identification of templates with macros and suggest conditionally choosing which part of a template is to be used.

But now our templates are no longer simple, no longer invariant. If we do this, we have lost the major advantage of templates, their simplicity. Thus a better answer would be to build simple templates, design a compiler using them, and then refine the compiler. Conditional templates are beginning to specify *how* rather than *what*. The latter is more valuable as a planning tool.

A second pass over the AMA code generated by the inefficient templates might eliminate redundant store-load pairs, and help with jump-to-jumps, and perhaps eliminate unnecessary loads in expression evaluation. This second pass, however, could know only the local structure, and would have to be conservative in its changes.

A simple method exists, however, that can alleviate some of these problems. It does not provide for the optimization of common sub-expressions, but can help in the expression evaluations and eliminate some of the store-load pairs. Where the source construct leads to an inefficient template, perhaps the source construct can be divided into two or more related constructs, thus rewriting the grammar of the source language. Each of these constructs would have its own template. The choice of templates is still unconditional, the source has not been changed, and some efficiency can be gained. This was done for assignment statements. The initial IL had one assignment statement construct only, for

mutiple assignments. The template for these was inefficient for an assignment to only one target. The assignment statments were divided into two classes: 'single' and 'plural'. The old template was assigned to 'plural', and a new one avoiding a redundant store-load was made for 'single'. The IL was not changed.

As an example of this solution, expressions could also be formally divided to avoid the unnecessary loading of a right-hand operand. Distinguish between the production used for an operation that has a variable or constant operand and the production used for an operation both of whose operands are sub-expressions. Each will have its own template. The template of the first kind can use the direct memory reference to the variable or the location holding the constant. The template for the second type will use the register reference of the old template.

This could be carried even further, to allow the immediate mode to be used in constant references.

Chapter V

THE CODE GENERATOR

5.1 A SHORT DESCRIPTION

5.1.1 Scope of Chapter

The compiler was divided into several modules, two of which, the ARM and the CG, generated code. The ARM is not covered in this thesis. The CG module did call the FE, AMA and ARM, but none of these needs to be documented other than as sources or targets of information. Thus only the CG, as I wrote it, will be described.[16]

5.1.2 Overview

The CG consisted of a parser, which called a set of semantic routines on recognition of syntactic structures. The semantic routines would sometimes generate code, calling the AMA for that purpose, and sometimes would modify various tables that described the variables or the code that had already been produced.

---

[16] Modifications subsequently introduced by others to my code may be described without special notice where they are corrections of bugs in my version. Where they are not corrections, they will not be mentioned.

The semantic routines also used various service routines within the CG, such as the register allocator. Semantic routines also called routines in the ARM to generate array-handling code, and in the FE to get the description of a constant in the source program.

Each major division of the CG will be given a short description in the rest of this chapter, but for details the reader is refered to the fourth appendix, where the FORTRAN code is described.

5.1.3 The Parser

The CG parses the IL source program with an LL(1) table-driven parser. The tables were generated, prior to slight modification to make them FORTRAN 'DATA' statements rather than PL/I initializations, by a program written by a graduate student here, Robert Keeler, who had left before we started the compiler.

The parser uses a service module to maintain the stack required by an LL(1) parse (see 5.1.5, where the stack module is covered).

The entries in the table encode the actions to be taken by the parser. If the parse cannot continue, the entry reveals the error, and gives a unique number for each kind of error. If there is no error, the entry determines the actions to be taken by the parser. The possible actions are

reading a new token, popping the top element off the grammar stack, and pushing the right-hand side of a production onto the grammar stack. More than one action may be specified. If a production's right-hand side is to be pushed, the table-entry encodes the number of the production. The sequence of popping the top element off the stack and pushing the right-hand side of a production is called applying the production.

If the table-entry says that no error has occurred, and some production may be applied, a semantic routine caller is called. It has the form of a gigantic 'case' statment (simulated in FORTRAN) and calls small subroutines for most of the possible productions. There is one case for each production in the IL grammar.

In some of these cases, a small subroutine is called to perform the associated semantic action, while for a few, the action is simple enough that it is performed within the case statement branch. Thus, on recognition of the beginning of a 'dood' construct, a routine is called to generate the beginning of the template, and another when the end of the construct is recognized.

5.1.4 Semantic Routines

These semantic routines communicate with each other by means of four stacks, one for each of the major divisions of

the routines: there is one for the 'do od' and 'if fi' constructs, one for expressions, one for the guards within an 'if fi' or 'do od', and one that helps in the generation of code to evaluate the 'cand' and 'cor' operations. Another stack serves the LL(1) parser, as mentioned before.

Some of the semantic routines will be sketched here; all are covered in the fourth appendix.

5.1.5 Stack Module

Five stacks in all are provided by the stack module, though some of them could be replaced by one stack. These five are separate for two reasons: some stacks could not be coalesced, as their combined usage is not LIFO, and clarity and maintainability are increased by having one stack for each function. Other LIFO elements in the compiler are contained within other modules, such as the symbol table, or the 'INDOOD' table that records the nesting depth of 'dood's within the nested blocks of the program.

5.1.6 Symbol Table Module

Variables in the IL are entered into a symbol table with their attributes. The symbol table and the functions associated with it form a separate sub-module which allows entry, retrieval, block entry and exit, inheritance from an enclosing block with new attributes, and dumping of the table as a debugging aid.

The PE also has a symbol table, used to translate source names into IL names. The CG's table holds the following attributes for each IL name: AM location that holds its value, initilzation status, accessibility status, initial scope, current scope, and type. The source name is not known.

5.1.7 Register Allocation Module

The other semantic routines call the register allocation module to allocate or release registers. The CG assumes there are at least three distinct registers, and possibly more (the number is available from the AM).

In all cases, a request must indicate the type of the register and register life must be LIFO. Before a register may be released, all registers allocated after its allocation must have been released. It is one of the most restrictive features of this module that both the compile-time and the run-time history must follow this rule. Both allocation and release must specify the same type. If these conditions are not met, erroneous code will be generated--the module cannot detect misuse.

This module is easily both the most individual and the most far-reaching in its effects, and thus deserves some discussion. I decided to make my register allocator LIFO in order to make it simple to write, and to make the allocation

of an arbitrary number of registers independent of the number, as well as independent of the constructs of the language and of the rest of the compiler. The LIFO restriction proved confining, as an unexpected part of the CG-ARM interface (see 7.1.3.2) . The awkwardness of this module is due to the fact that it was the first one I wrote, and that it was designed before the CG-ARM interface was fixed in its final form.

The initial version of the allocator generated code to stack the registers' contents at run time. An improved version, saving time and space, generated loads and stores from AM temporaries, whereas the current version tries to reuse these temporaries as well.

5.1.8 AMA Interface Buffer Routines

To enable the compiler to continue to check the source code after an error has been detected and code can no longer be generated, the CG and the ARM call the AMA through buffer routines that call the AMA only if an error flag is not set. They can also print the AM code if a debug flag is set.

5.2 POSSIBLE IMPROVMENTS TO THE CG

FORTRAN space usage could be much improved by linearizing the parse tables in the FE and the CG. But 'optimizing' FORTRAN code is beyond the scope of this thesis.

Because the CG treats each language construct in isolation, there is no easy way to modify it to reduce greatly the amount of AM-code used by linking constructs together or sharing code between constructs. The CG was not designed to generate the most efficient AM code, but to implement the templates. In view of the time constraints, efforts to make the CG more efficient than it was already (due to its clear and straight-forward design) would have been counter-productive.

There is, however, plenty of room for the maintainers of the compiler to improve it. This might best be achieved by reducing the amount of AM storage used for variables and instructions. A few suggestions follow.

5.2.1 Better Templates

Improvement of the templates so that they would require fewer AM variables and generate fewer instructions is relatively easy in concept, but I can think of no good candidates other than the expression templates. They have already been discussed in the chapter on templates (4.6).

5.2.2 Reuse of AM storage

First, the scalar variables and the CG's temporaries can be reused. DPL arrays are already being reused, as they are simulated by the ARM rather than being AM arrays. The code for this reuse has been written and tested.[17] It is

described the fourth appendix. While not assuming any particular characteristic of the IL variables, such as block structure, it does assume that the IL is not in error when it says that a certain variable may be freed.

5.2.3 Removal of Indeterminacy

Second, the shuffling of the arrays of pointers to guards that introduces some uncertainty (to the writer of the DPL program, not to the analyzer of the compiler) into the evaluation of guards within a 'do od' or 'if fi' requires a subroutine call, and the existence of a subroutine, let alone the probable extra real machine code that must be generated to perform all the indirect references.[18] This AM sub-routine and the calls to it could be eliminated, if the loss of non-sequential and changing guard evaluation was not felt to be damaging to the semantics of DPL.

5.2.4 Miscellaneous

Improvements of the kind often called 'optimizations' such as elimination of jumps-to-jumps and redundant store-load pairs across construct boundaries may be very difficult to achieve given the current structure of the CG. If the AMA allowed reading and rewriting of its contents, a second pass

---

[17] It is being used in the latest version in the register allocator, but not in the rest of the CG.

[18] But this is precisely what the CG is not supposed to know, and shows that old habits of programmers die hard.

over the code could improve it. It would take considerable redesign to make the evaluation of common sub-expressions take place only once.

Superficial improvements can be made to the FORTRAN itself in the CG which would make it more readable. These are not listed here, as they are not related to the structure of the CG. They are in the fourth appendix, where they apply.

5.2.5 Following the Rules

Considerable work is also needed to make the DPL compiler follow all the rules of DPL that relate to initialization, type, and inheritance. Dijkstra requires that corresponding variables and expressions in multiple assignment statements have the same type. The CG does not check for type compatibility in multiple assigments or in array initializations, though where this would be done is noted in the appendix. More recent versions of the compiler do perform some of these checks.

Further, if one branch of an 'if fi' has an initialization for a variable, all the branches must have an initialization for the same variable, no matter how deeply buried in enclosed blocks or other constructs. Because initialization in 'if fi's must occur in parallel, two initializations for the same variable are not tagged as an error by the CG.

The refinement of the initialization rule enforcement would bring our version of DPL closer to that used in A Discipline of Programming. Other differences between the language our compiler implements and the language we tried to implement are listed in 6.2.

Chapter VI

THE DPL COMPILER AND ITS GOALS

The DPL compiler has been in use by students since the beginning of 1980, no member of the team that designed or wrote it is now involved with the compiler, and so it is fair to ask how well the project met its goals.

It is only fair to mention that considerable work has been done on the compiler since I left the project, resulting in much improvement, according to Dr. Parnas [Parnas, personal communication, 1980]. This chapter details the state of the compiler for the early fall of 1979.

6.1 RUNNING DPL PROGRAMS AS WRITTEN

As yet, there has not been an effort to run all of the programs in Discipline of Programming though this would be an obvious way to check the compiler's fulfillment of our major goal. Certain differences between the DPL of the book and our DPL exist, but will probably not cause problems. The goal, however, is not met.

6.2 DIFFERENCES BETWEEN OUR DPL AND THE BOOK'S VERSION

There are three major differences: the addition of input and output (IO), the fact that the binary boolean operations have been implemented in a manner that changes their semantics, and the inability of the compiler to detect certain errors of initialization status. None of these invalidates correct programs, but the third difference makes it harder for students to find errors in incorrect ones.

6.2.1 Input and Output

Dijkstra describes no IO for his notation. We added input and output in a way we feel follows the spirit of the language: the user, as the outermost block, may specify the initialization of three input and three output arrays. Each of the two (input and output) consists of one array of each type (integer, boolean and character). At the end of the program's execution the output arrays are printed.

Rather than make the user write all the DPL for this outer block, only the initial values of the input arrays need be specified. Input is normally performed with the 'lopop' operation on an input array, and output with the 'hiext' operation on an output array, but the arrays may be accessed in any legal fashion.[19] The ability to perform IO

---

[19] 'Lopop' removes the element with the lowest index from the array, shortening it, 'hiext' adds a new element above the one with the highest index, increasing the size of the array.

should not affect the running of Dijkstra's programs.

6.2.2 CAND rather than AND

Our compiler does not implement the logical operations 'and' and 'or'. Rather than evaluate both operands in all cases, the second operand is not evaluated if the first determines the result (for example, 'true or x' is always 'true'). We call these new operations 'cand' and 'cor'. I do not know why the decision was made to implement these operations rather than the logical ones. Given the lack of side-effects to the evaluation of expressions, this should not be detrimental either.

6.2.3 Unenforced Rules

The rules about the use of initialized and uninitialized variables that the compiler does not enforce should have no effect on the running of correct programs. The lack contributes, however, to my judgement on the compiler in the last section of this chapter, and to the compiler's suitability for student use. The errors that the compiler does not detect and their effect on student use of the compiler are detailed in the section on student-orientation (5.4.3).

6.2.4 Exponentiation

Our compiler allows the binary operation of exponentiation, which Dijkstra does not use. There is no effect on any of the goals.

6.3 PORTABILITY

Until another computer facility uses our compiler and its own version of the AM to build a DPL compiler, the portability of the compiler can only be the subject of an educated guess.

James D. George, the designer of the AM and the only person who has written an AM to date, is pessimistic about the likelihood that other installations will write AM's to transport the compiler. In characteristically cautious terms, he says:

> I could not categorically deny two possible assertions about the Abstract Machine:
>
> 1. It is too dificult to implement.
> 2. It does not exploit target-machine power well.
>
> [George, 1979, p. 130]

For the implementer elsewhere the choice is probably between using our compiler and writing a new AM and writing his or her own compiler, using a high-level language and such tools as parser generators, pre-written symbol table routines, or already-existing assemblers. The deciding factor is likely to be the expected effort.

George, in his thesis [George, 1979, p. 132], estimates that his version of the AM on the IBM 360 took him two man-

months, not including the time it took to learn about the AM or the IBM 360. His AM does not perform run-time type-checking, a deviation from the definition, and he was, of course, very familiar with the AM.

In contrast, Gary Bishop of this department wrote a DPL compiler as a course project, and estimates that he spent one man-month on it [Bishop, 1980, personal communication]. His version has certain faults: there is no character data type, and there are no error messages for type and initialization constraints, and none for run-time errors. Twice the effort, two man-months, would certainly improve the product. They might not improve it enough to be used as a student compiler.

As the estimated time for transporting the compiler is the same as the time for writing one's own compiler, I do not think it likely that the compiler will be transported elsewhere. When the time needed to understand the AM is added to the time needed to write an AM, transporting would seem the more difficult task. Our goal of easy portability is only partially achieved.

6.4 STUDENT ORIENTATION

When one considers the compiler as a tool for naive students, the DPL compiler has certain obvious problems but provides enough information to its user that both the kind

of error and its location can usually be found. These faults are described in the rest of this section as is the question of our compiler's diagnostic aids (again, the reader is reminded that some of the faults have been corrected in versions of the compiler later than the one this thesis treats).

6.4.1 Variable Names in Messages

Many messages do not refer to a source variable, for example 'missing semicolon' or 'guard expression is not boolean'. Those that do are more informative if they can specify which variable is the guilty one; thus '"x" is uninitialized and cannot be used in an expression' is a more useful message for a student than 'uninitialized variable in an expression'. The DPL compiler messages generated by the CG are not in terms of the source variable, but like the second example, only name the problem.

The error is, however, detected and pinpointed to a particular statement (but see the next subsection) and, in most cases, will be enough to enable the student to correct his or her program. The major weakness is that in long expressions or in multiple assignments, there may be several possible variables at fault. A suggested solution to this problem is given in the next chapter (7.2.5) .

6.4.2 Line and Statement Numbering

When the FE reads a DPL program, it prints a copy. This copy has line numbers. In the IL produced by the FE, there are statement numbers, which are used by the CG to pinpoint the statement in which an error has been detected. Unfortunately, these two sets of numbers are not the same: a statement may be split into several lines, or a line may contain several statements.

A solution to the confusion produced by a message referring to statement ten, which is printed on line eight, is proposed in the next chapter (7.2.5) .

6.4.3 Undetected Errors

There are rules about the use of initialized and uninitialized variables that the compiler cannot enforce, as it does not detect a mistake by the programmer. Possible mistakes are failing to initialize a variable in all branches of an 'if fi', failing to make all the intializations for a variable of the same type, failing to match the types of array-intialization elements with the declared type of the array, and mismatching the types of left and right-hand sides in a multiple assignment. Further, the compiler does not check the type of an array index in all array operations, and the incorrect type (for example, a boolean index value) will cause the program to fail.

If a student makes one of these errors it will become visible only at run-time, and only if the erroneous code is executed. The symptoms will be a message from the AM, complaining about invalid register types, and the program will halt. There will be no output printed from the output arrays. This will be very little help to the student programmer.

Suggested solutions to the problems of initialization are in the following chapter (7.2.6) and, in more detail, in the documentation describing the CG.

6.4.4 Lack of Diagnostic Aids

The PL/C compiler, used at UNC-CH in the introductory programming course, does more than echo and number the source program and list the output: it produces a number of aids, such as depth-of-nesting numbering beside each line, an identifier cross-reference, and, in the event of an abnormal end, a short trace, in terms of the PL/C statment numbers. The DPL compiler produces none of these aids.

We deliberately did not include such aids in our initial design. The goal of student-orientation might seem to require that our compiler help the student by producing such aids. Though there is (and was) disagreement on this topic, it is not the task of this thesis to do more than note that we did what we had planned in the way of diagnostic aids.

6.5 MODULARITY AND INFORMATION HIDING

Our desire to make full use of the techniques of division into modules and of information-hiding was fulfilled, and very fruitful: when the time came to put our separately-written modules together, there were no problems at the interfaces. This is a remarkable testimonial.

Further, when a totally new team of programmers was adding major improvements, they were made with relative ease and almost no change to the inter-module interfaces [Parnas, personal communication, 1980]. This, too, is an indication of the benefits of these techniques.

There are parts of the compiler where we could have used the techniques more thoroughly but did not. Had we done so, I am certain the compiler would have been simpler to write and debug.

Thus the FE has an interface in two parts to the CG, but only needed one. The first and major portion is the IL, available through a function call to NEXTOK in the FE. The second is the constant-holder part of the FE, which returns the length and elements of constants through two functions. For a more unified interface, the constants could have been embedded in the IL, and the responsibility for the character-to-boolean or character-to-integer conversion could have been given to the FE.

Within the CG, the only module whose interior I know, there are tables that would have been hidden within a submodule if I had known earlier that I would need them. An example is a table INDOOD, recording the depth of 'do od' nesting for each block: this would have been part of the stack module, but my need for it did not become apparent until after the stack module was finished and its interface broadcast to all the CG in the form of a common block with the stack names. Rather than try to find all the references to the common block, I added a new table. This, and a other few instances in the CG, had no impact on the rest of the CG or the other modules.

6.6 PROVABILITY

One version of the templates has been proven correct: so far as the compiler realizes the templates, it is correct. The effort to prove the templates uncovered many errors in the templates (mostly in register usage) that would have been very hard to detect otherwise. Though proving the templates correct was not a goal, it has been very useful.

6.7 DELIVERY DATE

We did not meet our delivery date. The compiler took more than twice as long as had been planned, twenty months rather than nine. Here we did not meet our goal.

6.8 EFFICIENCY

Though efficiency was never a goal of the project, it is proper to ask whether or not the DPL compiler is so inefficient that it will not be used. This is also relevent to the possibility of transporting the compiler elsewhere: if it is inefficient, that is all the more reason to write one's own compiler.

As an example, the same algorithm was run on the PL/C and the DPL compilers, with the following compilers.[20] The statistics are given beneath the program listings.[21]

I did not try to favor one or the other language, but PL/C may have an advantage in better input and output operations. The difference is still immense: almost two orders of magnitude more time to execute, more than two orders of magnitude more compilation time, and almost three times the memory use. Another measure is the cost, where the ratio favors the PL/C job: eighteen cents to three dollars and three cents, a factor of seventeen. While the DPL compiler has other goals than efficiency, it is only fair to remember that the PL/C compiler is made larger by its extensive error-correction capacity.

---

[20] Note the use of the IO operations on the special arrays 'iinput' and 'ioutput'.

[21] Due to the accounting method used, the totals are not always the sums of the component entries.

DPL	PL/C
begin	TRY: PROCEDURE OPTIONS (MAIN);
glovar iinput, ioutput;	DELCARE (I, X) FIXED BINARY;
privar i, x;	
x vir int, iinput:lopop;	GET LIST (X);
i vir int := 1;	I = 1;
do i <= 20 ->	DO WHILE (I <= 20);
i, x := i + 1, x + 1	X = X + 1;
	I = I + 1;
od;	END;
ioutput: hiext (x)	PUT LIST (X);
end	END TRY;
CPU time, seconds	
compile: 8.8	.06
run: .8	.01
total: 9.6	.1
other time: 14.4	3.4
memory in K: 446	160
disk accesses: 762	146

The efficiency of the AM is not the problem, though the real machine code has about 4.2 IBM 360 instructions for each AM instruction (George, p.112). The generated code is not the problem, as it is not overly inefficient. The problem lies in the size of the compiler and the many passes over the source program.

6.9 JUDGEMENT ON THE DPL COMPILER

The compiler is late, it fails to enforce rules that are basic to the language, it is inefficient and its error messages can be less than useful. Clearly, there is a lot left

to do before the DPL compiler can be judged a success. Neither can it be dismissed as a failure: our compiler has made good on some of its goals, and partially achieved the others. It lies in the area of the 'low pass' or 'gentleman's C': too good to throw away, but disappointing.

Chapter VII

CONCLUSIONS, RECOMMENDATIONS

7.1 WHY THINGS WENT WRONG

There were two reasons for the low quality and late delivery of our compiler: an inexperienced team and poor communications within the team. Our optimism lead to an early predicted delivery date, making the delivery seem later than it would have been if we had appreciated the difficulties ahead. Each of these three aspects of what went wrong will be discussed.

7.1.1 Beginner's Luck and its Lack

None of the graduate students in the project had written a compiler before nor had any had formal instruction in the subject. Though eager to start, we were slowed by our need to learn how to do what we were doing.

Thus the parse method initially chosen for the CG was recursive descent, despite the difficulty in simulating recursion in FORTRAN, as I did not know there were parse-table generators available, or that other techniques of parsing would still allow me to generate the whole of a template in one (or two) subroutines. This decision stood until Dr.

Mehdi Jazayeri persuaded me to use another, more practical method, when the project was over a year old.

We did not worry about the size of the compiler until the end of the project: then we were amazed. A little forethought would have allowed the compiler to operate as several passes, rather than having the FE, CG, AMA, and ARM all present at the same time. More experienced people might have foreseen this.

We also underestimated the amount of time it would take us to learn how to use the AMA and AM, to use the module specification technique of traces [Bartussek and Parnas, 1977], and to learn to use FORTRAN. Our initial optimism became pessimism as the project continued long past our projected delivery date. Neither was justified by the actual state of the project.

7.1.2 Intermediate Language Improvements

More experienced people might also have noticed that we were proposing to parse the source language twice, once in the FE and once in the CG, but that the intermediate form was not designed to make the CG's parser small and simple. If the IL had been only slightly modified, with each construct unambiguously flagged at its beginning, the CG's parse tables would be much reduced.

An example may make this clear. It is only in the sixth element of the IL statement that an array initialization and a multiple assignment whose first identifier is being initialized are distinct. To make both the problem and the solution clearer, I give the IL. Here the items in MAJUSCULES are IL tokens, those in miniscules are variables, and non-terminals are in <brackets>. The IL is shown only for the beginning of the construct.

Intermediate Language

```
<array initialization> ::=
ASSIGN name MARK1 INITZN <type> ARRYSN . . .

<multiple assignment>  ::=
ASSIGN name MARK1 INITZN <type> MARK2 another-name . . .
```

Suggested Form

```
<array initialization> ::=
ARRAYINIT name <type> . . . .

<multiple assignment>  ::=
MULTSN name INITZN <type> . . .
```

7.1.3 You Meant THAT?

We experienced problems communicating decisions and definitions during the project. These problems were due to the choice of Dijkstra's book as the language definition, to misunderstandings, and to lack of distribution of information within the team. If a language definition had been written in advance, by one person, much time would have been saved that we spent looking in the book (for example, to

find whether or not there were character variables), or in disputation.

7.1.3.1 Problems with the Book

A Discipline of Programming had no index, did not gather all the definition of the notation in one place, and did not include a formal grammar. There was no definition of input or output. It was hard to use as a reference.

7.1.3.2 Misunderstandings

There were a few misunderstandings that did not get straightened out until after they had led to redundant or awkward code. Both the FE and the CG thought that checking variable inheritance was part of their function: thus both do it. Though the AM is designed to trap on detection of an attempt to divide by zero, or to take a residue modulo zero, the CG generates tests before division or residue operations.

An example of awkwardness due to inadequate interface specification is the conflict of the CG's register allocation method with the ARM's interface to the CG: to the CG, registers are arbitrary and allocated by a submodule, to the ARM, registers are named and their use is defined by the programmer. Thus before each call to the ARM generated by the CG all registers must be allocated, and all freed after the call, to preserve LIFO usage of registers. This means extra loads and stores for parameters to the ARM.

We had not realized that register allocation would be part of an interface using registers, nor that an interface involving shared storage locations might be less likely to create problems.

7.1.3.3 Unbroadcast News

Certain decisions (for example, the form of the IO we would add to DPL) did not get successfully broadcast from the decider to the rest of the team. Further, questions about overall design did not get quick answers from the leader.

7.1.4 Conceptual Integrity and Idle Hands

In my opinion, if one person had extracted a formal definition of DPL from Dijkstra's book, designed an extension for IO, and sketched the secrets and tasks of the modules within the compiler, the result would have been less wasted time in the beginning, and a better understanding by the rest of the team of both the task at hand and how long it would take.

The Mythical Man Month [Brooks, 1975, p 47] discusses the temptation to put idle hands to work, even when the current task is best done by those already working. This may be the reason that the compiler was designed by a committee (Dr. Parnas, Dr. Wagner, Jim George and me), rather than by one person alone.

7.2 HOW TO FIX THINGS

The following few, short suggested solutions may not cover all the faults of the DPL compiler. If implemented, however, they should improve it. The first suggestion is aimed at the past, rather than the future. The documentation for the CG covers some of the fixes in more detail (with code, in a few cases). Solutions are described under the head of the goal they foster. Some faults have already been solved in more recent versions of the compiler.

7.2.1 Shortening the Time Needed

The two largest modules could have been split, reducing the size of each sub-module. These modules could have been written by different people, thus shortening the over-all time. In both cases, the internal interface was specified by the writer of the large module even though it was not split.

If the CG had been divided into two sub-modules, one of which parsed and provided utilities such as the symbol table and the stack module, while the other generated code and allocated registers, the CG might be easier to maintain, and all the rules might have been enforced.

If the AMA and the AM had been written by two different people, two AM's could have been written, and the portability of the compiler tested to a greater extent.

7.2.2 Student-Orientation

The line-number versus statement-number problem could be best solved by each statement, would be available to the source-echoing submodule. COMMON, initially set to zero and incremented at the beginning of each statement, would be available to the source-echoing submodule. It would label each echoed line with the number of the first statement on it.

The FE should also be extended to include a routine that would print the DPL name of a variable in the IL when called by the CG. This name is already available in the FE's character holder submodule. The names of variables could then be printed in error messages.

7.2.3 Implementing DPL

To check the type of expressions in a multiple assignment, within an array initialization, or in array operations, the routine CTYPOK, described in the CG's documentation should be added to the CG. It is called after an expression has been generated, and compares the type of the expession to the type expected. Both of these are in COMMON blocks already.

Checking that initializations proceed in parallel in the branches of an 'if fi' is harder. I can think of two possible avenues to a solution, one at compile-time and one at run-time.

During compilation, a list of the variables that must be initialized within a particular 'if fi' could be built for each 'if fi'. The list would be built while translating the first branch, and checked within the others. If a new variable was initialized, or if one of the listed variables was not initialized, that would be an error. As the 'if fi's can be nested, there would be a set of lists, and this might require a lot of code and storage.

These lists would be checked on initialization of a variable, as an expansion of the routine CASSN6.

If the compiler left an AM version of the symbol table, with type and initialization information, a run-time routine called for each initialization before the assignment was made could check that those assignments that were made were correct. This approach, however, could not detect required initializations that were not made.

A simpler solution is to change the message printed by the AM when the generated code trys to load or store an object of a different type than the referenced area of storage contains. This would usually imply an initialization error in the source. If the message were "Probable error in initialization of variable in 'if fi' ", rather than "Reg type err", the other solutions would not be needed.

7.2.4 Efficiency and Portability

I recommend removing the trap mechanism from the definition of the AM.[22] This would have two good effects: the AM will be easier to implement, it is simpler, and the real machine code produced will be smaller and thus run faster. What error-checking is needed can provided by code generated by the CG, or by the real machine.

7.2.5 Epilog

The fact that I now feel I could do a much better job is indicative of the amount I learnt as a member of the project team. I hope that our experience will help others to do better without the slowness and pain of learning by experience.

---

[22] Dr. Parnas disagrees with this recommendation.

Appendix A

ATTRIBUTE GRAMMAR

This attribute grammar serves three functions: it presents a BNF grammar of the IL that was the source language produced by the FE for the CG; it lists the templates which were used to translate constructs into AM code; and it formally specifies the translation, and thus the the CG, without over-specification. Two things should be noted, however: the specification is not complete, and the very important question of register management is ignored.

The ARM and the code it generates are ignored, as is the code generated to allow input and output.

Further, what is more important from a formal point of view, the translation of source symbols into AM memory locations is unspecified. From a practical point of view, however, it is unimportant, as any function that maps elements of the source into a part of AM memory without overlaps is enough. This is only true if the source is correct, and all variable references follow the scope rules. But templates assume correct input. For a full specification we would need a more powerful notation.

How information is save for later use is not included. This makes some productions seem useless: they may have no translations in code, but are none the less important.

Register allocation is determined by the implementor, and is not properly part of a template. The templates may express requirements for the register allocation algorithm to meet. I have tried to show these by the use of specific register names: e.g. R1 is register one, Ra or Rb an arbitrary register. Still, the templates say only 'with result in R1', and do not say how this is to be done.

A.1 NOTATION USED

Each rule in the IL grammar is numbered, and the corresponding translation equivalence is below it. Code that is emitted and IL terminals are in CAPITALS, IL non-terminals are in <bracketed miniscules>, indicies in unbracketed miniscules, while restrictions and explanitory material are in underlined English, to advoid a complicated new formalism. Braces [ ] are added for clarity. Thus a grammatical rule in IL might be:

<non-terminal> ::= one or more [terminal(s)]

.us one or more [<nonterminal(s)>] .

The sequence of generation in the translations is from top to bottom. Initilizations of arrays are given in a shortened form:

Arrayname (*) <-- list of values.

This stands for the long sequence of SETs and NULLs after a DSA. Otherwise, the reservation of storage is explicit. For explainations of the AMA code, see Appendix C. As it is there, the type-specifying suffix 'B', 'C', 'F', 'P', or 'L' is replaced by '_' to avoid useless repetition of the same template.

'TEMP' is always an unused scalar location of the appropriate type.

A.2 THE GRAMMAR

1. <program> ::= <block>
   t(<program>) =

```
                JUMPL   BEGIN, NR, D
MSG1 (*)        <-- 'ABORT STMT EXECUTED'
ABORT:          WRITN   PRINTER, MSG1, NR, M, 25, NR, M
                HALTN
MSG2 (*)        <-- 'NO GUARD OF IFFI TRUE'
IFBORT:         WRITN   PRINTER, MSG2, NR, M, 22, NR, M
                HALTN
MSG3 (*)        <-- 'ATTEMPT TO DIVIDE BY ZERO'
ZERDIV:         WRITN   PRINTER, MSG3, NR, M, 26, NR, M
                HALTN
MSG4 (*)        <-- 'ATTEMPT TO TAKE ZERO MODULO'
ZERMOD:         WRITN   PRINTER, MSG4, NR, M, 27, NR, M
                HALTN
SHUFL:          STOP    R1, RETURN, NR, D
                LOADP   R2, SNAME, NR, D
                LOADF   R3, 0, NR, M
                LOADP   R1, R3, R2, BX
                STOP    R1, TEMP, NR, D
                LOADF   R3, 1 NR, D
LOOP:           CJCF    R3, SNUM, NR, I, GE, DONE, NR, D
                LOADP   R1, R3, R2, BX
                SUBF    R3, 1, NR, M
                STOP    R1, R3, R2, BX
                ADDF    R3, 2, NR, M
DONE:           LOADP   R1, TEMP, NR, D
                LOADF   R3, SNUM, NR, I
                STOP    R1, R3, R2, BX
                JUMPL   RETURN, NR, I
RETURN          DS      POI
SNAME           DS      POI
SNUM            DS      POI
TRUE            DC      '1'B
FALSE           DC      '0'B
LINENUM         DS      FXD
BEGIN:          t(<block>)
                HALTN
```

2. <block> ::= <ldecl> <lstmtln> <lfreeables>
   t(<block>) = t(<lstmtln>)

3. <ldecl> ::= <decl> <ldecl>

4. <ldecl> ::= ENDLST

5. <decl> ::= GLOVAR | VIRVAR | PRIVAR | GLOCON | VIRCON | PRICON

6. <lstmtln> ::= <stmtln> <lstmtln>
   t(<lstmtln>) = t(<stmtln>) t(<lstmtln>)

7. <lstmtln> ::= ENDLST
   t(<lstmtln>) =

8. <lfreeables> ::= i <lfreeable>

```
9. <lfreeables> ::=  ENDLST

10 <stmtln> ::= LINUM i <stmt>
                    (where  0 < i )

   t(<stmtln>) =    MMMF  i, NR, M, LINENUM, NR, D
                    t(<stmt>)

11. <stmt> ::= PGM <block>
   t(<stmt>) = t(<block>)

12. <stmt> ::= DOOD one or more (n) of
                  [ <expression i> <lstmtln i> ] ENDLST
    t(<stmt>) =          JUMPL  START, NR, D
     for each of the n sets <expression i> <lstmtln i>
     [      GUARDi:      STOP   R1, SAVE, NR, D
                    t(<expression i>)
                              into register Ra
                         CJCB   Ra, TRUE, NR, D, NE, SAVE, NR, I
                    t(<lstmtln i>)
                         JUMPL  SUCCESS, NR, D
                                                          ]

            GUARDS <-- GUARD1, GUARD2, ... GUARDn
            GNUM       DC     n
            SAVE       DS     POI
            SUCCESS:
            START:     MMMP   GUARDS, NR, M, SNAME, NR, D
                       MMMF   GNUM, NR, M, SNUM, NR, D
                       JLKL   SHUFL, NR, D
                       LOADF  Rb, [n - 1], NR, M
            LOOP:      JLKL   R1, GUARDS, Ri, XI
                       SUBF   Rb, 1, NR, M
                       CJCF   Rb, 0, NR, M, GE, LOOP, NR, D

                            Where 'b' does not equal '1'


13. <stmt> ::= IFFI one or more (n) of
                  [ <expression i> <lstmtln i> ] ENDLST
    t(<stmt>) =          JUMPL  START, NR, D
     for each of the n sets <expression i> <lstmtln i>
     [      GUARDi:      STOP   R1, SAVE, NR, D
                    t(<expression i>)
                              into register Ra
                         CJCB   Ra, TRUE, NR, D, NE, SAVE, NR, I
                    t(<lstmtln i>)
                         JUMPL  SUCCESS, NR, D
                                                          ]
            GNUM       DS     n
            SAVE       DS     POI
            GUARDS <-- GUARD1, GUARD2, ... GUARDn
            START:     MMMP   GUARDS, NR, M, SNAME, NR, D
                       MMMF   GNUM, NR, M, SNUM, NR, D
```

```
                    JLKL    SHUFL, NR, D
                    LOADF   Rb, [n - 1], NR, M
LOOP:               JLKL    R1, GUARDS, Ri, XI
                    SUBF    Rb, 1, NR, M
                    CJCF    Rb, 0, NR, M, GE, LOOP, NR, D
                    JUMPL   IFBORT, NR, D
SUCCESS:
```

Where 'b' does not equal '1'

14. <stmt> ::= SKIP
    t(<stmt>) =

15. <stmt> ::= ABORT
    t(<stmt>) = JUMPL ABORT, NR, D

16. <stmt> ::= ASSIGN i ARRYOP HIREM
    t(<stmt>) = LOADF R2, AMNM, NR, M
    call array manager for HIREM

17. <stmt> ::= ASSIGN i ARRYOP LOREM
    t(<stmt>) = LOADF R2, AMNM, NR, M
    call array manager for LOREM

18. <stmt> ::= ASSIGN i ARRYOP HIEXT <expression>
    t(<stmt>) = t(<expression>)
    with the result in Ra
    STO_ Ra, TEMP, NR, D
    LOADF R2, i, NR, M
    LOADP R3, TEMP, NR, M
    call array manager for HIEXT

19. <stmt> ::= ASSIGN i ARRYOP LOEXT <expression>
    t(<stmt>) = t(<expression>)
    with the result in Ra
    STOF Ra, TEMP, NR, D
    LOADF R2, i, NR, M
    LOADP R3, TEMP, NR, M
    call array manager for LOEXT

20. <stmt> ::= ASSIGN i ARRYOP SHIFT <expression>
    t(<stmt>) = t(<expression>)
    with the result in Ra
    STOF Ra, TEMP, NR, D
    LOADF R2, i, NR, M
    LOADP R3, TEMP, NR, M
    call array manager for SHIFT

21. <stmt> ::= ASSIGN i ARRYOP SWAP <expression 1> <expression 2>
    t(<stmt>) = t(<expression 1>)
    with the result in R1
    t(<expression 2>)

```
                                    with the result in R3
                          LOADF  R2, i, NR, M
                                    call array manager for SWAP

22. <stmt> ::= ASSIGN i ARRYOP ALT <expression 1> <expression 2>
    t(<stmt>) =  t(<expression 1>)
                                    with the result in R1
                 t(<expression 2>)
                                    with the result in R3
                          LOADF  R2, i, NR, M
                                    call array manager for ALT

23. <stmt> ::= ASSIGN i MARK1 <init> SIMPSN <expression>
    t(<stmt>) =  t(<expression>)
                                    with the result in Ra
                          STO_    Ra, t(i), NR, D

24. <stmt> ::= ASSIGN i MARK1 <init> MARK2 j POP HIPOP
    t(<stmt>) =           LOADF  R2, j, NR, M
                                    call array manager for HIPOP
                          STO_   R3, t(i), NR, D

25. <stmt> ::= ASSIGN i MARK1 <init> MARK2 j POP LOPOP
    t(<stmt>) =           LOADF  R2, j, NR, M
                                    call array manager for LOPOP
                          STO_   R3, t(i), NR, D

26. <stmt> ::= ASSIGN i1 MARK1 <init> MARK2 MULTSN <init>
                                    one or more indices i2, i3,  .
                                    . i(n - 1), in
                                    one or more [<expression j>]

    t(<stmt>) = for each <expression j>,
                [    t(<expression j>)
                                    into R3
                          STO_   R3, TEMPj, NR, D    ]
                                    for each i(j), from j = n to 1
                [         LOAD   R3, TEMPj, NR, D
                          STO_   R3, t(i(j)), NR, D  ]

26. <stmt> ::= MULTSN i MARK1 <init> ARRYSN
                                    one or more [ j(k) ] ENDLST
    t(<stmt>) =         LOADF  Ra, 0, NR, D
                                    for each k
              [         MMM_   t( j(k) ), NR, M, ARRAY_, NR, Ra
                        ADDF   Ra, 1, NR, M                      ]
                                    call array manager for ini-
                                    tialization

28. <init> ::= ACTIVE | INITZN <type>
    t(<init>) =

29. <expression> ::= <postfix> ENDLST
    t(<expression>) = t(<postfix>)
```

20. <postfix> ::= SIMPLV i
    t(<postfix>) = LOAD_ Ra, t(i), NR, D

31. <postfix> ::= PROPERTY i <array property>
    <array property> ::= DOM | LOB | HIB | HIGH | LOW
    t(<postfix)> = LOADF R2, i, NR, M
                   call Array Manager for array property named. DOM, LOB, and HIB all return their value in R1, HIGH and LOW in R3.
    t(<array property>) =

32. <postfix> ::= CONSTN <type> i
    t(<postfix>) =
                   call Front End to get value of constant named 'i' of the type specified
                   LOAD_ Ra, constant, NR, M

33. <type> ::= INT | BOOL | CHAR

34. <postfix> ::= SUBSCR i <expression>
    t(<postfix>) = t(<expression>) into register a
                   call Array Manager to get the 'expression' th element of the array named 'i'.
                   which puts the value into register three

35. <postfix> ::= <postfix> NEG
    t(<postfix>) = t(<postfix>) into register i
                   LNEGF Ri, NV, Ri, BM

36. <postfix> ::= <postfix> NOT
    t(<postfix>) = t(<postfix>) into register i
                   LNOTB Ri, NV, Ri, BM

37. <postfix> ::= <postfix> ABS
    t(<postfix>) = t(<postfix>) into register i
                   CJCF   Ri, 0, NR, D, GE, NOFLIP, NR, D
                   LNEGF Ri, NV, Ri, BM
          NOFLIP:

38. <postfix> ::= <postfix 1> <postfix 2> PLUS
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>) into register j (j and i distinct)
                   ADDF   Ri, NV, Rj, BM

39. <postfix> ::= <postfix 1> <postfix 2> MINUS
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>) into register j (j and i distinct)
                   SUBF   Ri, NV, Rj, BM

30. <postfix> ::= <postfix 1> <postfix 2> TIMES

```
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>)  into register j (j and i
                   distinct)
                  MULFF  Ri, NV, Rj, BM

41. <postfix> ::= <postfix 1> <postfix 2> DIVIDE
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>)  into register j (j and i
                   distinct)
                  CJCF   Rj, 0, NR, M, EQ, ZERDIV, NR, D
                  DIVF   Ri, NV, Rj, BM

42. <postfix> ::= <postfix 1> <postfix 2> MODULO
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>)  into register j (j and i
                   distinct)
                  CJCF   Rj, 0, NR, M, EQ, ZERMOD, NR, D
                  MODF   Ri, NV, Rj, BM

43. <postfix> ::= <postfix 1> <postfix 2> CAND
    t(<postfix>) = t(<postfix 1>) into register i
                  CJCB   Ri, TRUE, NR, D, NE, FAIL, NE, D
                    t(<postfix 2>) into register i (the same
                    one)
           FAIL:

44. <postfix> ::= <postfix 1> <postfix 2> COR
    t(<postfix>) = t(<postfix 1>) into register i
                  CJCB   Ri, TRUE, NR, D, EQ, PASS, NR, D
                    t(<postfix 2>) into register i (the same
                    one)
           PASS:

45. <postfix> ::= <postfix 1> <postfix 2> LESS
    t(<postfix>) = t(<postfix 1>) into register i
                    t(<postfix 2>)  into register  j (i and j
                    distinct)
                  CJC_   Ri, NV, Rj, BM, LT, YES, NR, D
                  LOADB  Rk, FALSE, NR, D
                  JUMPL  DONE, NR, D
           YES:   LOADB  Rk, TRUE, NR, D
           DONE:
46. <postfix> ::= <postfix 1> <postfix 2> EQUAL
    t(<postfix>) = t(<postfix 1>) into register i
                    t(<postfix 2>)  into register  j (i and j
                    distinct)
                  CJC_   Ri, NV, Rj, BM, EQ, YES, NR, D
                  LOADB  Rk, FALSE, NR, D
                  JUMPL  DONE, NR, D
           YES:   LOADB  Rk, TRUE, NR, D
           DONE:

47. <postfix> ::= <postfix 1> <postfix 2> MORE
    t(<postfix>) = t(<postfix 1>) into register i
```

```
                    t(<postfix 2>)  into register  j (i and j
                    distinct)
                  CJC_    Ri, NV, Rj, BM, GT, YES, NR, D
                  LOADB   Rk, FALSE, NR, D
                  JUMPL   DONE, NR, D
          YES:    LOADB   Rk, TRUE, NR, D
          DONE:
```

48. <postfix> ::= <postfix 1> <postfix 2> NEQUAL
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>) into register j (i and j
                   distinct)

```
                  CJC_    Ri, NV, Rj, BM, NE, YES, NR, D
                  LOADB   Rk, FALSE, NR, D
                  JUMPL   DONE, NR, D
          YES:    LOADB   Rk, TRUE, NR, D
          DONE:
```

49. <postfix> ::= <postfix 1> <postfix 2> NLESS
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>) into register j (i and j
                   distinct)

```
                  CJC_    Ri, NV, Rj, BM, GE, YES, NR, D
                  LOADB   Rk, FALSE, NR, D
                  JUMPL   DONE, NR, D
          YES:    LOADB   Rk, TRUE, NR, D
          DONE:
```

50. <postfix> ::= <postfix 1> <postfix 2> NMORE
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>) into register j (i and j
                   distinct)

```
                  CJC_    Ri, NV, Rj, BM, LE, YES, NR, D
                  LOADB   Rk, FALSE, NR, D
                  JUMPL   DONE, NR, D
          YES:    LOADB   Rk, TRUE, NR, D
          DONE:
```

51. <postfix> ::= <postfix 1> <postfix 2> POWER
    t(<postfix>) = t(<postfix 1>) into register i
                   t(<postfix 2>) into register j (i and j
                   distinct)

```
                  STOF    Ri, TEMP, NR, D
                  LOADF   Ri, 0, NR, M
          LOOP:   CJCF    Rj, 0, NR, M, LE, DONE, NR, D
                  MULIF   Ri, TEMP, NR, D
                  SUBF    Rj, 1, NR, M
                  JUMPL   LOOP, NR, D
          DONE:
```

Appendix B

INTERMEDIATE LANGUAGE MNEMONICS

In actual fact, the Intermediate Language was numeric: the values of the mnemonics used in the attribute grammar are given in the table below.

ABORT	12	HIEXT	57	NEQUAL	-38
ABS	-27	HIGH	51	NLESS	-39
ACTIVE	21	HIPOP	62	NMORE	-40
ALT	61	HIREM	55	NOT	-36
ARRAY	0	IFFI	9	PGM	7
ARRYOP	13	INITZN	20	PLUS	-28
ARRYSN	15	INT	52	POP	18
ASSIGN	10	LESS	-35	POWER	-41
BOOL	53	LINUM	99	PRICON	6
CAND	-33	LOB	48	PRIVAR	3
CHAR	54	LOEXT	58	PROPERTY	44
CONSTN	43	LOPOP	63	SCALAR	0
COR	-34	LOREM	56	SHIFT	59
DIVIDE	-31	LOW	50	SIMPLV	46
DOM	49	MARK1	14	SIMPSN	17
DOOD	8	MARK2	16	SKIP	11
ENDLST	0	MINUS	-29	SUBSCR	45
EQUAL	-36	MODULO	-32	SWAP	60
GLOCON	4	MORE	-37	TIMES	-30
GLOVAR	1	MULTSN	19	VIRCON	5
HIB	47	NEG	-25	VIRVAR	2

There was also an end-of-file marker that the FE would generate at the end of the IL. It was not part of the IL. Its value was -100.

Note that operations were negative, and all others were positive, with the exception of SCALAR and ARRAY, which were dropped from the IL. They are included here for historical interest.

Appendix C

SEMANTICS OF THE AMA AND THE AM

The AM is a simple machine. It has a number of registers, an area of memory that can be addressed, and is very like any simple machine. The innovative parts were not used by the CG, though the ARM did, as noted below, ask the AM to allocate and free memory. The templates in the first appendix are in a pseudo-assembly language, but there should be no difficulty in understanding that through the machine code described here, as the use of instruction labels is obvious, and the pseudo-operations DS, and DC should also be clear. I use these in preference to the AMA calls to avoid the question of relative timing of the GETL and the SET_, but will give a table of equivalences (ignoring timing, using the generic- type convention).

Pseudo-Assembly			AMA
name	DS	type	GETL (name, type, SCA, F)
name	DC	value	GETL (name, type, SCA, F) SET_ (name, type, value, F)
(values are written as 'a'B--BIT, 'A'--CHR, n--FXD)			
label:			GETT (label, F) TAGI (label, F)

An AM program is a sequence of instructions and a set of named memory and instruction locations. These locations are referenced in the instructions by three parameters (or portions of the instruction, if you will). Their meaning is defined in the first table. The meanings of the instructions are defined in the next three tables.

C.1 ADDRESSING OPTIONS

For each addressing method used (and others offered but not used) I will describe the effective address using the 'contents of' operation, written 'C( )', and the indexing operation, written '[ ]'. 'Name' refers to a location in AM memory, either an scalar, or an array, in which case an index is needed to complete the reference (except for pointing to an array, when the use of an index is forbidden).

The address is described in three parameters: L, R, and F. L is a name or a register designation, R is a register designation, and F is one of the following flags:

At one point we considered another flag setting, for the case where both the name of an array and the index into it were known at compile-time. It would have been convenient

Flag	Meaning	Comments
B	C(R)	register holds a name
D	L	a name
M	no address, operand is	immediate with value L
I	C(L) = C(D)	indirection
X	L[C(R)]	index in register
BM	R	register name
BX	C(R)[C(L)]	index and name in registers
BY	C(R)[L]	index is immediate
IB	C(C(R)) = C(B)	
IX	C(L[C(R)]) = C(X)	
MX	L+C(R) A must be integer	
BIX	C(C(R)[C(L)])) = C(BX)	
BIY	C(C(R)[L])) = C(BY)	

for such references, as the code generator would not have to generate code to load one of elements into a register, nor would a register have to be allocated to hold either the index or the name of the array. This 'Y' flag (definition R[L]) was finially rejected on the grounds that in a real machine one of two would have to be loaded into a register, and that asking the porter to perform this action would be making the task too dificult. It would only have been used by the ARM.

C.2 THE INSTRUCTION SET

The instructions are executed sequentially, starting at the first one. This order of execution is modified by some operations as noted below, and by trapping. Each instruc-

tion has an associated address, by which the transfer of execution can be described. The transfers associated with the trapping mechanism will be treated in the next section.

As in the templates, I will use an underscore (_) to replace one of the type-specifying suffixes. Thus rather than describe LOADB, LOADC, LOADF, and so on, I will describe LOAD_. Where there are constraints on the types that may be suffixed to the generic command, the command will be listed with the various type suffixes. Thus 'ADDF' rather than 'ADD_' is listed, as only FXD data may be added.

The three parameters that specify an effective address will be replaced by 'EA'. Where an effective operand is expected (that is, where immediate mode is legal) I will use 'EO'. Except for immediate mode, it is always true that C(EA) ≡ EO. The operation of copying from right to left is written as '<-'. Thus the first line of the table below is to be read 'the contents of R become a copy of the effective operand'. Note that LCFF and LFFC only allow conversions from the CHARs '0123456789' to the FXDs 1, 2, 3, 4, 5, 6, 7, 8, 9 and vice versa. '¬' is 'not'.

Operation	Meaning
LOAD_ R, EO	C(R) <- EO
STO_ R, EA	C(EA) <- C(R)
JUMPL EA	execution continues at EA
JLKL R, EA	C(R) <- the address of the next instruction execution continues at EA
CJC_ R, EO, F, EA	F is a comparison, one of: {<, =, >,]<,]=,]>}; if { C(R) F EO } is true, execution continues at EA, otherwise as usual
MMM_ EO1, EA2	C(EA2) <- EO1
ANDB R, EO	C(R) <- C(R) |and| EO
ORB R, EO	C(R) <- C(R) |or| EO
XORB R, EO	C(R) <- C(R)]= EO
LNOTB R, EO	C(R) <-] EO
SRB R, EO	C(R) <- shift C(R) EO bits right
SLB R, EO	C(R) <- shift C(R) EO bits left
ROTRB R, EO	C(R) <- rotate C(R) EO (mod 16) bits right
ADDF R, EO	C(R) <- C(R) + EO
SUBF R, EO	C(R) <- C(R) - EO
MULIF R, EO	C(R) <- C(R) * EO upper half of product
MULFF R, EO	C(R) <- C(R) * EO lower half of product
DIVF R, EO	C(R) <- C(R) / EO
MODF R, EO	C(R) <- C(R) mod EO
LNEGF R, EO	C(R) <- - EO
LCFF R, EO	C(R) <- integer version of the character EO
LFFC R, EO	C(R) <- character version the integer EO
DRLL R, EA	C(R) <- C(R) - 1; then if the new content is not zero continue execution at EA
DMLF R, EA	C(EA) <- C(EA) - 1; then if the new content is not zero continue execution at the location C(R)
HALTN	execution stops
NOPN	nothing happens

C.3 TRAPPING MECHANISM

The trapping mechanism works in the following way: the AM detects an error, of class 'i', it continues execution at the location pointed to by the 'i'th element of a special array TRAPS, initialized by the commands that can terminate a trap: ARM (most are just TRETN). There are three ways to leave a trap routine: note that JUMPL will not work when a trap has occurred, and one of the instructions below must be executed to return to normal operation.

TCON	continue execution at the instruction within which the trap occured.
TRETN	continue execution at the instruction after the one within which the trap occured.
TENDL EA	continue execution at EA.

The CG did not use the AM trapping mechanism. Explicit tests for zero were performed before division and modulo, rather than using the AM trap for zero-division and zero-modulo. All traps that occur are thus errors of the compiler's, not the user's.

C.4 INSTRUCTIONS NOT USED BY THE CG

Other instructions that the CG did not use were designed for the Array Manager (ARM). These are described here for the sake of completeness only. To aid understanding of these instructions, the reader must know that untyped AM storage is held as an array of elements called LCMs.

```
MSA_   E01, E02, E03, E04, E05
            for i = 0 to E05 do
                C( E01 )[ E02 + i] <- C( E03 )[ E04 + i]
                end
NLCMF EA    C( EA ) <- the number of LCM units
ALLOM E01, E02, E03, EA  allocate E02 units of LCM
                         starting at index E01 to
                         to be type E03. The new name
                         of this array is L;
                         C( EA ) <- L
FREEM E01, E02, E03, EA  free the array named EA, of
                         type E03. This corresponds
                         to the E02 units of LCM
                         starting at E01 that was
                         previously allocated by an
                         ALLOM call
```

Appendix D

DOCUMENTATION OF THE CG

In this appendix the Code Generator will be documented. This is a description of the generator I wrote, not of the abstract generator that is partially specified in Appendix A. I plan to describe the CG in the following manner: first the routines which do not generate code, starting with the service routines, going on to the parsing routines, and the non-code producing semantic routines, second those routines that do generate code, starting with the ones that do not call others, and thence to the more complicated expresssion generators.

This documentation assumes a familiarity with FORTRAN, and common data structures such as stacks, and tables. I will not specify ownership of a common block by a caller which owns the block only to assure that the values in the common block do not change. As the Astract Machine language elements were given hidden values, which were held by variables in common blocks accessable to the CG, I will use only the variable names, not the values. All variables are INTEGERs or LOGICALs, and explictly declared in the code. These declarations will also not be specified.

As a guide to myself and other maintainers of the code generator, I put a 'G' on the names of routines that generated code, and a 'C' on the other's names.

After the thumbnail description of each module I show how to call it, and the meaning of the parameters. Parameters that are changed or set by a routine are prefixed with an asterix (*).

For each routine, all errors detected are listed, with their associated numbers (for reporting to CERRPT), and the sub-modules it calls.

The AMA routines are called indirectly, so that code generation may be traced by setting a flag: thus GETT is replaced in the CG by GHETT, which calls GETT, and may print the values of the parameters. The buffer routines have an 'H' added to the corresponding AMA name.

D.1 ERROR REPORTING AND DETECTION

When a routine detectes an error, it branches to its end, by-passing all non-error code, to a label number of nine thousand or more. The error flag passed in is set to a

unique number specifying the error, and CERRPT (C-ERror-RePorT) is called, with the flag as its sole parameter. This routine prints a message giving the current line number, the current token (for debugging purposes - but possibly a later version of the Front End could retrieve the source token that gave rise to the IL token) and a message describing the error. Implementation errors, such as stack overflow, are printed with leading equal signs, others with leading stars. CERRPT calls no routines.

CERRPT (error-number)

Errors detected: unknown error code (no number)

Calls: nothing

D.2 AMA BUFFER ROUTINES

The CG and the ARM use these routines, rather than calling the AMA directly. Each routine checks the flag in the common block /CGMORE/. If it is true, the AMA is called. If not, it is not (a newer version also has a flag that controls printing the parameters to the AMA, in the same manner). These routines are exactly like the AMA's, except for the names, and thus will not have their parameters listed. They are: GHETL, GHETT, THAGI, SHETB, SHETC, SHETF, SHETP, DHSA, KODE5, KODE9, and KODE16.

They detect no errors.

D.3 THE STACKS

The stack routines provide LIFO stacks for the rest of the compiler. There are five stacks named via common block variables in /CSTACK/, and five functions: initialization, pop, push, top and empty, provided by CSTKIN (C-STacK-INitialize), CPOP, CPUSH, CTOP and CEMPTY. The stacks are held in a common named /CLIFO/, along with their depth in MXDEEP, and an array of pointers to the top element of each stack. Overflow, underflow and invalid stack name are dectected, as is calling CTOP of an empty stack. The current depth is thirty, but as MXDEEP is used everywhere, only the declarations in CSTKIN and the initialization of MXDEEP need be changed to change the depths. All stacks have the same depth, but this probably does not reflect their real depth in use. None of these routines call any other module, except CERRPT.

CSTKIN (*error flag)

Errors detected: none

CPOP (name of stack, *error flag)

Errors detected: invalid stack name 81

stack underflow 82

Calls: nothing

CPUSH (name of stack, item to push, *error flag)

Errors detected: invalid stack name 79

stack overflow 80

Calls: nothing

CTOP (name of stack, item at top, *error flag)

Errors detected: invalid stack name 83

stack empty 84

Calls: nothing

CEMPTY (name of stack, *true if empty, *error flag)

Errors detected: invalid stack name 86

Calls: nothing

D.4 THE PARSER

The parser is a LL(1) table-driven parser, using the stack named GRAM (GRAMmar stack). The parser gets new tokens from the IL by calling NXXTOK (a buffer routine for the FE's NEXTOK which may print new tokens as a debugging aid) and then calls GSEMAN to perform semantic actions. Errors detected in the IL are reported to CGEPH, whereupon parsing is stopped. CGEPH is a FE routine, and is not described here.

Given a token and a non-terminal on the top of the stack, the token's type is translated into an index, and it and the non-terminal are used to look up an action in the parse table. GSEMAN is called for actions that are not errors, and is passed the number of the production in the IL grammar, the current non-terminal on the stack, the translated index for the token's type, and the token. An error flag is also passed. If the action specified requires the application of a production (the replacement of the current non-terminal by a string of terminals and non-terminals) then CREPLC is called and passed the production number and an error flag.

A peculiarity of the parse is the necessity to accept arbitrary numbers as variable names. There is a token type 'arbitrary number', but the incoming token that is accepted as such is not arbitrary, as far as the parse is concerned, but a specific number. Thus arbitrary numbers must be translated into this special number. It must be known when an abitrary number is expected, so that the action table row for the token 'arbitrary number' may be referenced. To do this, a list of the non-terminals that can be on the top of the stack when the token expected is an arbitrary number is in a common block /CGARBN/ (CG-ARBitrary-Number). Another common block, /CPHTAB/ (C-PHront end-Table) holds an array that allows translation of the tokens that are not 'arbitrary number' into token types and hence indices into

the parse action table. The parse action table is in the common block /GPARTB/ (G-PARse-TaBle).

CREPLC (C-REPLaCement of stack elements) uses a production table in the common block /GPROTB/ (G-PROduction-TaBle) and the stack commands to replace the top non-terminal by the set of elements in the table specified by the production number. Two other items in the common block guard against requests to replace elements according to a non-existent production.

GPARSE (*error flag)

Errors detected: FE error — error flag + 1 (sent to FE only)

Calls: stacks, NXXTOK, CREPLC, GSEMAN, CGEPH

CREPLC (production number, *error flag)

Errors detected: invalid production number — 1

Calls: stacks

D.5 THE SYMBOL TABLE

The symbol table module provides six functions: initialization, entry of a symbol with attributes, inspection of a symbol's attributes, dumping of the table, block entry,

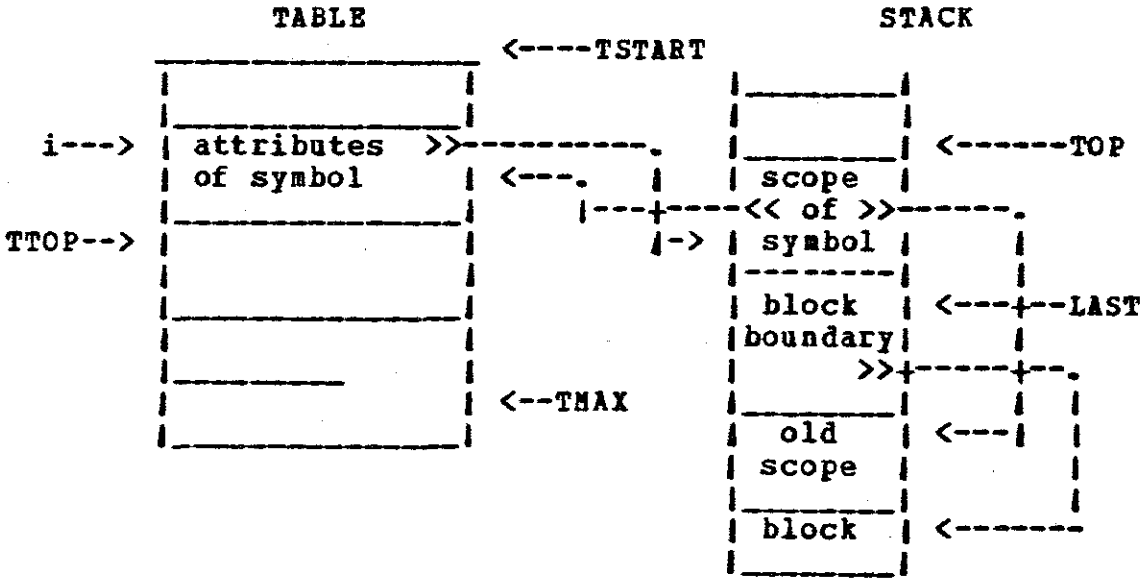
block exit, and inheritance of a symbol from the next outermost block with a new scope. The last three assume a block-structured language, and may need to be changed if DPL is changed, but the module design should make this easy.

The table is in the common block /CGSTAB/, along with various pointers into the table, and a stack to help handle inheritance. The items in this common block are described below.

As the symbols are guaranteed to be positive integers in sequence from some initial positive integer, it is easy to map them into indices (1, 2, 3, ... n) in the table. TSTART, initially set to -100, is the offset to be subtracted from the symbols to give the index. The table is of limited size--currently only two hundred symbols can be accomodated. The upper limit for a symbol is given in TMAX, which is equal to TSTART + 200. The last new symbol will have the largest index, and the next will go in the table element with an index one larger. This next index is held in TTOP.

Each symbol has seven attributes, the last of which is a pointer into the stack of scopes. TOP points to the next free element on the stack, LAST to a position on the stack corresponding to the last entry into a block. The stack,

called STACK, can hold one hundred entries. Each entry has three parts: a pointer (index) into the table, a scope, and a pointer (index) to an entry further down in the stack. The null pointer is represented by zero. Thus a symbol has a table entry that points to a scope entry in the stack, and a chain leads downwards from each scope entry to entries for the scope of that symbol in an enclosing block. The boundaries between blocks are entries in the stack, too, linked to other block boundaries downwards, but only the last field is used: the first two are zeros.



'i' is the index (symbol - TSTART) of the most recently entered symbol.

At the bottom of the stack is a block boundary that is linked to itself. This is put in during initialization.

The common block /CGCSYM/ holds constants meaning 'uninitialized variable', 'initialized variable', 'accessable in this block', and 'inaccessable in this block': VIR, NONVIR, YES, NO. Actual values are 10, 11, 12, and 13.

None of these routines call any others (except for CERRPT)

CSINIT (C-Symbol table-INITialize) initializes the table and the stack, and gives values to the elements of /CGCSYM/. CSNTR (C-Symbol table-eNTeR) enters attributes for a symbol, setting TSTART and TMAX if the symbol is the first one, and creates an entry for the symbol on the stack if this is the first time attributes have been entered for it. CSNSPC (C-Symbol table-iNSPeCt) retrieves attributes, getting the scope from the topmost stack entry for this symbol. CSIN and CSOUT (C-Symbol table-IN, C-Symbol table-OUT) are called on block entry and exit, respectively. CSIN adds a block boundary marker to the stack, updates LAST, and makes all symbols in the next outermost block inaccessable. CSOUT makes all symbols in this block inaccessable, removes the entries for them from the stack, setting their pointers in the table to the next entry in the chain downwards, removes the block boundary, and then makes all entries in this block, which was the next outermost block, accessable. CSNHRT (C-Symbol

table-iNHeRiT) adds a new entry to the stack for a given symbol if it is in the next outermost block. CSNHRT does not check whether the scopes or types allow inheritance. CSDUMP (C-Symbol table-DUMP) prints the attributes and current scope for all symbols that have been entered in the table.

CSINIT (*error flag)

Errors dected: none

CSNTR (symbol, abstract machine name, initial scope, initialization status, type, array or scalar status, accessability, *error flag)

Errors detected: invalid symbol 29

CSNSPC (symbol, *abstract machine name, *current scope, *initializtion status, *type, *array or scalar status, *error flag)

Errors detected: invalid symbol 30

no entry for symbol 31

CSIN (*error flag)

Errors detected: stack overflow 32

CSOUT (*error flag)

Errors detected: none

CSNHRT (symbol, new scope, *error flag)

Errors detected: invalid symbol 33

uninheritable symbol 34

stack overflow 35

CSDUMP (*error flag)

Errors detected: none

D.6 MEMORY MANAGEMENT

In the initial design, there was no re-use of memory. When a new symbol was encountered, a unique location in AM-memory was assigned to it, using the AMA call GETL (*memory name, AM type, AM sort, *error flag). For arrays, as the array manager used the symbol as a name, this memory location was thrown away and not used. In any case, once a block had been left, there was no recovery of the AM-memory assigned to the variables in that block.

At the end of each block the FE provides a list of dead variables, variables which never again will be referenced. For each of these symbols, CFREE (C-FREE) is called. If the symbol was an array, the ARM routine for recovering memory is called by CFREE. There exists, however, code in CFREE (as comments) that calls a a routine for recovering scalar storage. The change in CFREE is minor: if an scalar, CFREES (C-FREE-Scalar) is called. To use these recovered locations,

all requests for memory names for scalar symbols is made through CNAMES (C-NAME-Scalar). CNAMES abd CFREES exist, but are not part of the released compiler. These two routines maintain four stacks, one for each AM type, putting a returned AM memory name on the appropiate stack, and returning a name when requested off the top of the appropiate stack. If the stack is empty, GETL is called.

The stack and its pointers are maintained in the common block /CGNAME/ K(4), STAC (30, 4). The K(i) are initialized to 1 in CINIT, described later.

```
CFREE (symbol name, *error flag)
  Errors detected: symbol not in table          12
                   unable to save scalar        13
                   unable to save array         14
  Calls: ARM, CERRPT

CFREES (AM name for symbol, AM type, *error flag)
  Errors detected: invalid type                 (no code yet)
  Calls: CERRPT

CNAMES (*AM name, AM type, *error flag)
  Errors detected: invalid type                 (no code yet)
  Calls: AMA, CERRPT
```

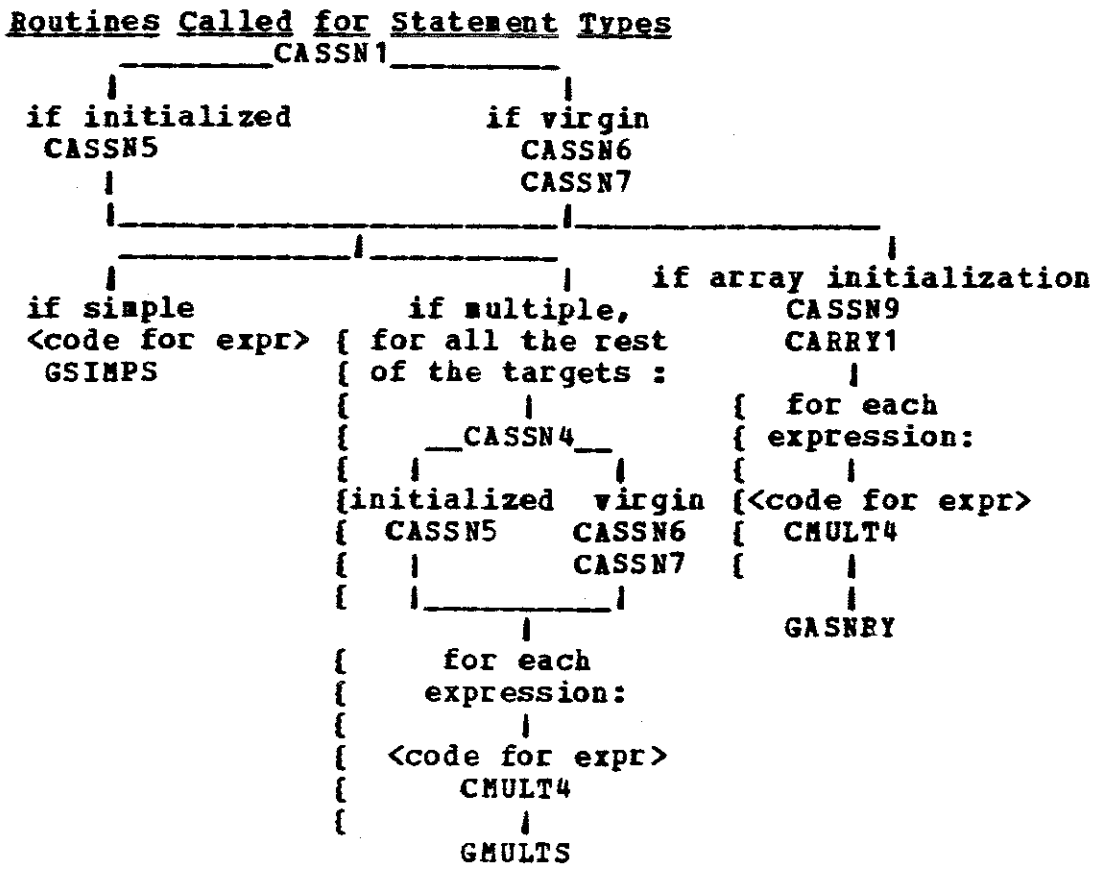
When these are compiled and linked in, they can be given error codes in sequence with those of the compiler.

D.7 CHECKING OF ASSIGNMENT TARGETS

A set of little routines check the entries for symbols that are assigned to in assignment statements. They inspect the symbol table entry for that symbol, save the AM memory name, and the type, and check that the initialization status is correct: though here much more checking could be done, as some of the language definitions were too hard to implement in this first version (a further discussion of this point is found in chapter six).

The diagram below shows the calling sequence of these little routines for the various kinds of assignment statements. It should help the understanding of the prose that follows.

When the first symbol is found in an assignment statement CASSN1 is called: it saves the symbol in a common block for the other routines in this module to use. /CGMULT/ NUM, INDX, TAB (30, 2) hold, respectively, the number of symbols encountered so far in the parse of a statement, the number of expressions encountered, and a table of two elements for each symbol: its name and type. CASSN1 sets the first two to one and zero, and records the symbol as the IL name.



If the assignment is not an initialization, then CASSN5 will be called next. It checks the symbol table entry for the symbol in TAB (NUM, 1) and puts its AM memory name and AM type in to TAB (NUM, 1 and 2), overwriting the IL name.[23] If this is an initialization, the symbol table is checked by CASSN6, but until the type is known no more can be done. Later, when the IL type for the symbol is known, CASSN7 is called, and it requests an AM memory name for the symbol,

---

[23] A possible improvement to this would be to make TAB three wide, rather than two. Both the IL and the AM name of a symbol would be available, and the use of TAB would be clearer.

records the name and AM type in TAB as CASSN5 does, and enters them in the symbol table.

Only later in an initialization would the compiler be able to tell whether or not it is an array that is being initialized. If it is, CASSN9 is called. It changes the symbol table entry to indicate that this symbol is an array, and throws away the new AM name.[24] As an array initialization continues, the initial index is met. It will be followed by a list of initial values. TAB (NUM + 1 through 30, 2) is set to reflect the expected types of expression to come: the initial index should be an integer, the rest of the same type as the array is being initialized to.

In a multiple assingment statement more target symbols follow. For each one, CASSN4 is called. It puts the symbol into the next location in TAB, and increments NUM. Thus the number of targets is counted. CASSN4 is thus like CASSN1, and it is also followed by either CASSN5, or by CASSN6 and CASSN7. Note that the compiler cannot handle the interleaving of array initializations with other assignments in a multiple assignment statement, though mixtures of initializations and non-initializations in a single statement are no trouble.

---

[24] See 'Memory Management'. Under certain circumstances, this AM location can be re-used.

The rest of this module generates some code. With it begins the documentation of the code-generation routines.

The right-hand side of an assignment or array-initialization statement is a list of one or more expressions. If there is only one, it is a 'simple' assignment, and once the single expression has been parsed and code has been generated for it, CSIMPS (C-SIMPle-aSsignment) is called. The compiler has described the location of the result of the expression in the common block /CTARGT/ (see the expression module for details). CSIMPS uses TAB (1, 2) and /CTARGT/ TYPE to check that the type of the target is the same as that of the result, and calls GASSN (G-ASSigN) to generate the code to make the assignment. If the result, as described in /CTARGT/, is not in a register, CSIMPS calls CMULT4 (C-MULTiple assignment) to generate code to load the result into register three, first.

In multiple assignments or array initializations, after the code for an evaluation of an expression has been generated, CMULT4 is called. It generates code to load the result into register three, and increments INDX in /CGMULT/ to count the number of expressions. This does not overwrite the previous contents of register three, as CMULT4 requests that register from the register allocater, which is described later. At the end of the list of expressions, another rou-

tine is called. If this is an array assignment, GASNRY (G-ASsigN-aRraY), is called, if a multiple assignment, GMULTS (G-MULTiple-aSsign). GMULTS checks that the number of expressions and the number of targets is the same, and then calls GASSN and the register allocater to make the assignments, using the AM memory names for the targets saved in /CGMULT/ TAB.

********** Fix-up for type-checking **********

It is in this module that type-checking of multiple-assignment (and array initialization) results should take place, but does not. To add it, after the production

<ENDEXPR> ::=

in GSEMAN, a routine CTYPOK (*error flag) should be called. It would be:

```
        CTYPOK (F)
        COMMON /CTARGT/ NAME, FLAG, TYPE
        COMMON /CGMULT/ NUM, INDX, TAB
        INTEGER NAME, FLAG, TYPE, NUM, INDX, TAB (30, 2), F
C
        IF (TYPE .NE. TAB (INDX, 2)) GO TO 9000
            F = 0
            RETURN
C
9000        F = new error number
            CALL CERRPT (F)
```

```
        RETURN
      END
```

This would be called after CMULT4 had incremented INDX.

********** End of fix-up **********

To return to the module as it exists, GASNRY puts the results in an AM array, puts the type, lower bound, and upper bound in another AM array, and calls the array manager. The two AM arrays are parameters for the array manager.

CASSN1 (IL name, *error flag)

Errors detected: none

Calls: nothing

CASSN5 (*error flag)

Errors reported:	unknown IL name	67
	inaccessable variable	68
	unitialized variable	69
	assignment to an array	70
	assignment to constant	71

Calls: symbol table, CERRPT

CASSN6 (*error flag)

Errors reported: unknown IL name 72
inaccessable variable 73
assignment to constant 74
initialization within dood construct 85

Calls: symbol table, CERRPT

CASSN7 (type, *error flag)

Errors reported: invalid type specified 75
unknown variable 76

Calls: symbol table, CFREES

CASSN9 (*error flag)

Errors reported: unknown variable 76

Calls: symbol table, CFREES

CARRY1 (*error flag)

Errors detected: none

Calls: nothing

CASSN4 (IL name, *error flag)

Errors detected: none

Calls: nothing

GMULTS (*error flag)

Errors reported: differing numbers of assignment

targets and results 77

Calls: register allocator, GASSN

CMULT4 (*error flag)

Errors reported: none

Calls: nothing

CSIMPS (*error flag)

Errors reported: target and expression
not same type 78

Calls: register allocator, GASSN

D.8 ASSIGNMENT CODE GENERATION

The generation of the code to perform the assignment is done by two little routines. This is really a sub-module of the previous one, which checks the assignment statement and generates code to put the result into the proper register(s).

GASSN is very simple: it is a case on the type of the result, generating a store command.

GASNRY generates code to put the results of the expressions in an array initialization into parameters for the

array manager subroutine that makes the assignment. The array values are put in an AM array of the right type, the initial index and calculated high index and type are put in another, and two registers are loaded with pointers to these arrays.

GASSN (name of variable, type of variable, register result is in, *error flag)

Errors detected: invalid type 2

Calls: AMA, CERRPT

GASNRY (*error flag)

Errors detected: invalid type 1

Calls: AMA, ARM, CERRPT

D.9 REGISTER ALLOCATION

I made a simplifying assumption about register usage: it would be LIFO, that is, no register could be freed until all registers allocated after it had been freed. This had to be true of both compile-time and run-time register usage. Thus registers tend to be allocated on entry into a construct, and freed on exit, and the name of the register that can be use is saved by the iffi and dood generation routines on their stack, DOIF.

The register manager routine is CREGMN (C-REGister-MaNager, the C- because, though it generates code, it is a compile-time service) and has five functions. Which function is desired is indicated by a parameter, whose value is hidden in the common block /CGREG/. The functions are named INIT, NAMED, ANY, REL and DIFF. The FORTRAN variables with those names hold the appropriate value. INIT initializes the register manager, REL frees a register, NAMED allocates a named register, ANY allocates a register and returns its name, and DIFF allocates a register different from the one named, and returns its name. Registers are always allocated as a certain type, and must be freeed as that type.

The common block /CGRMN/ holds the data structures that the manager uses. STATRG (i) holds the depth of use of register 'i'. Each time it is allocated, the depth is incremented, each time it is freed, it is decremented. The depth cannot be negative. TYPERG holds several stacks, one for each register. TYPERG (i, j) holds the type of the 'j'th deep allocation of register 'i'. Thus TYPERG (STATRG (i), i) holds the current type of register 'i'. The old contents of a register that is going to be allocated are stacked. In the old version of CREGMN, these stacks are in the AM, and code is generated to push and pop. In the newer version, the old contents are stored in an AM scalar, whose name is then stacked. These stacks are simulated in FORTRAN, not AM. As

all register usage is assumed to be LIFO (last-in-first-out) both at run-time and during compilation, either will work.

Because the different functions have different parameter semantics, the calling format is listed five times.

CREGMN (INIT, don't care, don't care, don't care, *error flag)

CREGMN (NAMED, name of register, type it will be, *error flag)

CREGMN (ANY, *name of register allocated, type it will be, *error flag)

CREGMN (REL, name of register, type it was, *error flag)

CREGMN (DIFF, *input as name of register to be different from and output as name allocated, type of register, *error flag)

Errors detected:	invalid operation	24
	invalid type	25
	non-LIFO use	26
	stack overflow (our error)	27
	invalid name of register	28

Calls: AMA, (if new version) CFREES, CNAMES

D.10 RUN-TIME SUPPORT

The Array Manager routines are not covered here, but the CG did generate the shuffling routine (to shuffle the pointers to guards in an iffi or dood) in GSHUFL, and the four traps: zero divide, zero modulo, no guard of an iffi true, and abort statement execution in GTRAP (now part of the ARM, with other trap routines, along with GABORT). GABORT generated the abort message trap code.

The Shuffle routine does a circular shift, the others each print a short message and halt.

The Fortran routine GSHUFL sets up three variables in a common block: /CSHUFL/ SHUFL, SNAME, and SNUM. SHUFL holds the AMA name of the shuffle routine, SNAME and SNUM are AMA scalar pointers to parameters for the routine. They are used by the dood and iffi-generation routines.

GTRAP has several common blocks that hold the AMA names of the various trap routines. They should be obvious: /CGABRT/ ABORT, /CBIN/ ZERDIV, ZERMOD, and /CIF/ IFBORT.

GSHUFL (*error flag)
Errors detected: none
Calls: AMA

GTRAP (*error flag)

Errors detected: none

Calls: AMA

GABORT (*error flag)

Errors detected: none

Calls: AMA

D.11 ARRAY-POP OPERATION

One of the new operations that DPL introduced was the array-pop, where a given array has its highest or lowest (by index) element removed, and a specified variable is set to the value of that element. GSEMAN calls GPOP to generate the code to perform the operation. GPOP will check that the array to be popped is indeed an array, and that it is initialized, and will allocate three registers and generate a call to the array manager. The element to be popped will be returned at run-time in register three (R3), and the assignment is made by a call to GASSN.

GPOP (hipop or lopop, *error flag)

Errors detected: variable not an array 62

array not initialized 63

command neither "hipop" nor "lopop" 64

Calls: ARM, AMA, register allocater

D.12 OTHER ARRAY OPERATIONS

DPL has many other array operations. they fall into two classes: those that alter an array, using the result of some expression, and those that return a value from an array description. The routine GARYOP does the first kind, the routine GARY the second. Those that return values can be elements of an expression; the others cannot.

GARYOP uses the name of the array (in /CGMULT/ TAB (NUM, 1)), checking that it is an aray, initialized, and accessable. It then performs a case statement (simulated in FORTRAN) based on the operation to generate code, allocate registers, call the appropriate ARM compile-time routine and free the registers again. For some array operations, there must be one or more parameters. If there is one, it is be found in the location described in /CTARGT/ (described in the expression section of this documentation: a name, a type, and a flag, which is .TRUE. if the name is a name of a register, .FALSE. if it is a memory location), or if there are two results, the first is in register three (R3), and the second is described in /CTARGT/. In some cases, the parameters to the run-time ARM must be stored in a temporary, and a pointer loaded.

********** TYPE-CHECKING NEEDED **********

Where there are two or more operands, only the last is described in /CTARGT/ for type-checking. To check the others, a routine must be called for the appropriate production in GSEMAN: CTYPOK is the obvious choice. Thus /CGMULT/ must be set up reflect the expected types: usually a FXD index and then either another index or an element of the array's base type.

/CTARGT/ should also be checked within GARYOP.

The table would be set up on seeing the operation, and checked by CTYPOK on completion of the expressions.
********** END OF NEEDED CHECKS **********

GARY is also basically a case statment after a check of the symbol table entry for the array. The calling sequence is simpler in GARY, as the parameters to the ARM are more similiar for these operations than for the first class. As GARY generates the code for part of an expression, it has parameters that describe part of an expression: an operation, and one or two operands.

GARYOP (operation to perform, *error flag)
Errors detected: reference to non-array or unitialized array 59
invalid array operation 60
invalid type of array 61
Calls: ARM, AMA, register allocater, CERRPT, symbol table

GARY (operation to perform, first operand name, flag (.TRUE. if first operand is in a register), type of first operand, second operand name, flag (.TRUE. if in a register), second operand type, *error flag)

Errors detected: array name result of caculation (in register) 86
array name not in symbol table 87
name not array name 88
invalid operation 69

Calls: ARM, AMA, symbol table, register allocater, CERRPT

D.13 THE DRIVER, INITIALIZATION AND IO

The driver owns all the common blocks of the CG, and all the common blocks of the other modules as well. It may vary from installation to installation. It will always do the following: call the AM to get a description of the AM's limitations, and put that information into common blocks; call initialization routines for all modules, including the CG; call the CG to generate code; and call the AM to finish generation and start excution or to save the AM program generated.

The CG's initialization routine is CINIT. It calls the initialization routines of the register allocater, the symbol table, and the stacks. It sets several constants,

including the names of the I/O vectors, creates the parameters for calling the ARM, and calls the routine that generates the run-time support routines (GSHUFL, and so on). Finially, it lables the beginning of the program.

Input and output (through-put) is done with six pre-defined arrays, two of each type. One set is input, and may be low-popped, the other is output, and may be high-extended. Other array operations on these arrays are legal. At the end of the program they are printed out by code generated by the routine CUTVEC (C-oUTput-Vector).

main routine
Errors detected: none
Calls: AM, ARM, PE, CG (CINIT and GPARSE)

CINIT (*error flag)
Errors detected: none
Calls: register allocater, stacks, symbol table, AMA,
GSHUFL, GTRAP (though a new version has merged
the old GTRAP with the ARM's traps)

CUTVEC (*error flag)
Errors detected: none
Calls: AMA, ARM, register allocater

D.14 SEMANTIC ACTION SELECTION

GSEMAN (G-SEMANtic-action) selects a semantic action when called by GPARSE. There are two kinds of production: those in which there is no replacement of the item on the top of the stack, and those in which the item is replaced by a string of non-terminals and terminals. In the second case, we have a production number to use in a case statement, but in the first we have to use the item on the top of the stack, as the production number is zero. The translation of the item on the top of the stack into a pseudo-production number is done via a computed go-to.

For most of the productions, nothing is done. For a few, like the one that gives the line number, code is generated, while for the rest, either some information is saved, or a subroutine is called, or both. Information is saved in the following common blocks:

/CGINDO/ THIS, OK, INDOOD (10): this records whether or not initialization is legal at this point in the parse. If we are in a do-od, it is not legal, unless we are also inside a block which itself is inside the do-od. At any time, we are in block number THIS, and INDOOD (THIS) equals OK if initilization is possible, that is, if we have not yet entered a do-od group, or have left the outermost do-od group in this block. INDOOD (THIS) is incremented for each

entry into a do-od, decremented for each exit. THIS is incremented for each entry into a block, decremented for each exit. If necessary, the size of INDOOD can be increased.

/CGSEM/ SCOPE, SNAME, STYPE: these record information to be used in the next few productions: SCOPE saves the scope for a list of declarations, STYPE and SNAME save the type and IL name of variable and constant references.

/CGTOK/ CTOK, LINE hold the current token and the current line number at compile-time.

/CGLINE/ LINENO is the AMA location in which the line number is stored at run-time.

GSEMAN (production number, non-terminal on top of GRAM stack,

class of token, token, *error flag)

Errors detected: invalid production

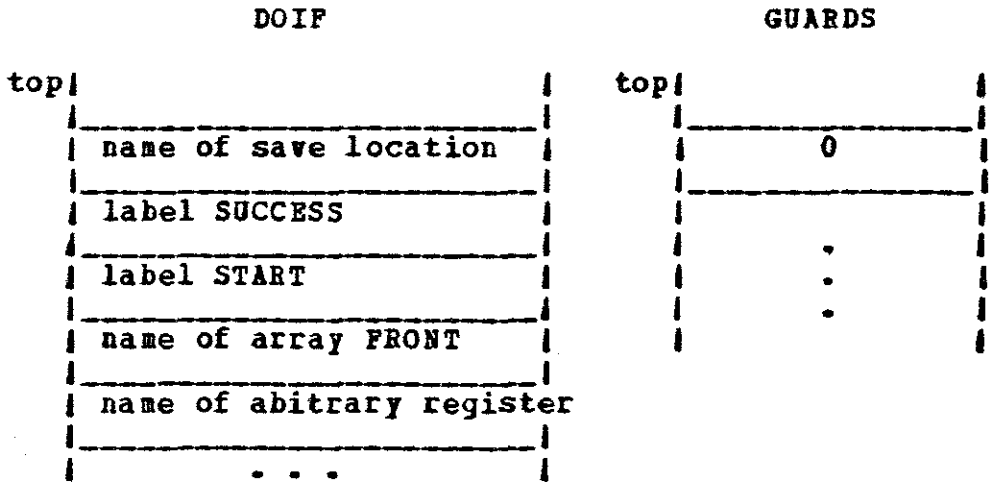
of an non-terminal 65

of a terminal 66

Calls: AMA, CUTVEC, symbol table, GDO, GOD, GIF, GFI, GABORT, CFREE, GCEND, GGUARD, CGDNSP, GARROW, CGENDGC, CASSN1, GARYOP, GARYEX, CMULT4, CASSN9, CASSN4, CSIMPS, GMULTS, CASSN6, CASSN5, CASSN7, CUNOP, CBINOP, CPAVAR, CONST, GEXPR, GSEXPR, CARRY1, GASNRY, GPOP, CERRPT

D.15 IFFI AND DOOD CONSTRUCTS

There are four routines that generate these constructs, and five more that generate the code for the guarded command set that is the heart of each construct. These routines communicate through the two stacks DOIF and GUARDS in the stack module. The first half of IFFI or DOOD gets labels and a save location for a pointer, allocates R1 and a arbitrary register, and codes a jump to the beginning of the control section (bypassing the code that will be generated from the guarded command set) and stack the following:



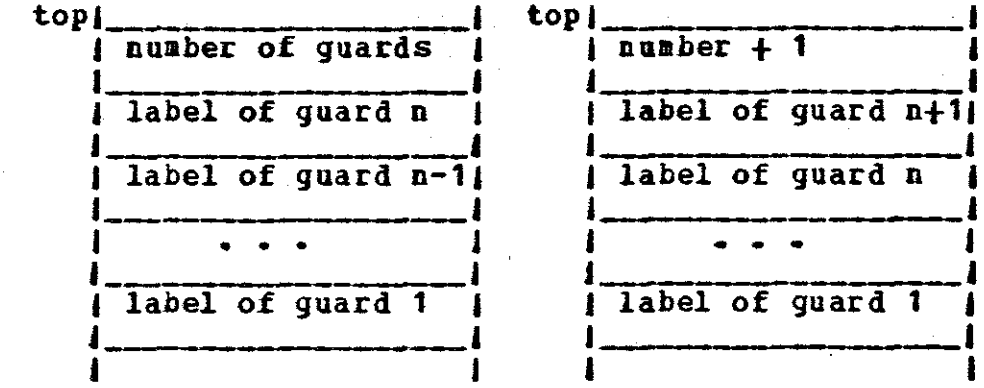
The second half of the IFFI or DOOD construct generates code to do the looping and branching that does the action of the construct. The arbitrary register is released, as is R1.

The calling sequence for the guarded command set routines is GGUARD to start a particular guard, GARROW to test the

guard, GENDGC at the end of a set of commands, and GGCEND at the end of the set of guards,before the iffi or dood control.

The stack GUARDS holds the number of guards encountered so far on the top, with the labels to the code that will evaluate the guards beneath it. GGUARD gets a new label, defines it, and stacks it, incrementing the count. CGDNSP (written by Dan Lanbeth) tests the result of the guarding expression to be it is of type 'BIT', and in register one, GARROW generates code to test the result, GENDGC to return to either the controlling loop (if a dood) or the next statement (if an iffi). GGCEND defines the array FRONT, in the stack DOIF, setting its elements to be pointers to the guard- evaluation routines.

GUARD stack: Before and after GGUARD



GDO and GOD do not directly affect the common block /CGINDO/ as might be expected. GSEMAN does the bookkeeping when it calls one or the other.

The routine CGINTO is called by CGDNSP. It generates code to put a result into a particular register, when that register is already allocated. If the result is already in a register, it assumes that only that register must be freed to make the target register accessable. In actuality, it is always called to put the result of a guard expression evaluation into R1 (register one).

********** INITIALIZATION CHECK NEEDED **********

I can't figure out how to check initializations within an 'iffi' to make sure they're duplicated in each branch. The code for the assignments does check to insure the types are the same.

The best I can suggest is building a list, for each 'iffi', of the variables that have to be initialized, and checking each branch to make sure they do. It would be tough. The lists would be built and checked in CASSN6.

********** END OF NEEDED CHECK **********

GDO (*error flag)

Errors detected: none

Calls: AMA, stack

GOD (*error flag)

Errors detected: none

Calls: AMA, stack

GIF (*error flag)

Errors detected: none

Calls: AMA, stack

GFI (*error flag)

Errors detected: none

Calls: AMA, stack

GGUARD (*error flag)

Errors detected: none

Calls: AMA, stack

CGDNSP (*error flag)

Errors detected: non-boolean guard result 87

Calls: CGINTO

GARROW (*error flag)

Errors detected: none

Calls: AMA, stacks

GENDGC (*error flag)

Errors detected: none

Calls: AMA, stack

GGCEND (*error flag)

Errors detected: none

Calls: AMA, stack

CGINTO (target register, *error flag)

Errors detected: none

Calls: AMA, register allocater

D.16 EXPRESSIONS

In the IL, expressions are in post-fix polish notation. Thus a simplified version of the CG's algorithm is:

The discussion below will give the name of the routine that does each part, and discuss the complications that arise in implementing the algorithm above. For example, the AMA provided unary operations only combined with loads: LNEGF and LNOTB. Thus the UNOP portion was modified as follows:

GSEMAN stacks all operators. When a variable referance has been parsed, GSEMAN calls GSEXPR to check it, and GSEXPR calls GEXPR. When a constant has been parsed (and the value retrieved from the FE by a call to CONST), GSEMAN calls GEXPR directly. GEXPR is the routine that performs most of the case statement in the algorithm above. It calls GUNOP

```
if {token is operand}
   then case {stack-top}
        UNOP --> {if {token is not register}
                     then {allocate register R
                           gen "load R, token"
                           token is now R}
                  gen "stack-top  token, token"
                  pop stack}
        BINOP -> push token
        else  -> {node := stack-top
                  pop stack
                 // stack-top must be a BINOP //
                  if {node is register}
                     then {gen "stack-top node, token"
                           token is now node}
                     else if {token is register and stack-top
                              is a commutative operation}
                          then gen "stack-top token, node"
                          else {if token is not register}
                               then {allocate R
                                     gen "load R,token"
                                     gen "stack-top R, node"
                                     token is R}
                               else {gen "store token, temp"
                                     gen "load token, node"
                                     gen "stack-top token, temp"
                  pop stack}
   esac
   else push token


UNOP ---> {if {token is not register}
              then {allocate register R
                    gen "unop R, token"
                    token is R}
              else gen "unop token, token"
         pop stack}
```

and GBINOP to generate code for the ordinary operations, and CXTRCT to generate code that will call the ARM to get an array element. The CG, you see, considers subscripting to be a binary operation. CPAVAR (written by Dan Lambeth) checks the array name, stacking it as an operand. It is called from GSEMAN.

The stack EXPR is used by this module, with the following encoding: each element has two stack elements describing it. An operator has a zero (0) stacked on top, an operand has its type stacked on top of it. If the type is positive, the operand is in a register, if it is negative, the operand is in an AM memory location. This is different from the normal system in the CG, where the type is always positive, and a flag describes whether or not the operand is in a register. GSEMAN does not know the encoding above.

CUNOP and CBINOP are called by GSEMAN to stack operands. Note the distinction: CUNOP, CBINOP stack operators, GUNOP, GBINOP generate code for the operations.

GEXPR calls various sub-routines to generate code, but there is one tricky thing that should be noted: 'cand' and 'cor' are not handled as ordinary binary operations, but decomposed into two unary operations. This had to be done, as the second operand may not be evaluated if the first gives the result of the 'cand' or 'cor'. Thus, when one or the other is seen by GEXPR, it calls a routine to generate some code for the first half, stacks a fictitious unary operator that will tell it to generate the second half on a later call. This accounts for the GCAND1 and GCAND2, GCOR1 and GCOR2, and the extra operators in the IL common block.

The 'cor' and 'cand' routines communicate through the stack called 'CANDS'. In it they stack a label in the code that is the end of the evaluation of the operation, and the name of the register that the result should be in.

When GEXPR returns, it has set some of its parameters to describe the result. If the stack is empty, this is the final result, if not, it is an intermediate result. Elements of the common block /CTARGT/ are set by GSEMAN to describe this result: its name, its type, and whether or not the name is a register name.

GBINOP is fairly complicated: it must ensure that registers are handled in LIFO fashion. It attempts to minimize register usage by not loading operands unless they must be loaded. This may require swapping the first and second operands. The result is always put into the "oldest" register. If two registers were used, the "younger" will be freed.

********** WARNING ! **********

GBINOP is the routine most likely to have been modified by other programmers since I left the project. What is written here may not be true anymore.

******** END OF WARNING**********

CONST calls the FE to get a value for a constant. If the constant is boolean, it can only be 'true' or 'false', so only the length (4 or 5) needs to be known. Note that CONST returns an AM name holding the constant, rather than the constant itself.

CONST calls CONVRT, which is the only machine-dependant routine in the CG. It translates IL characters into AM characters. Care should be taken when transporting the CG that CONVRT will still work.

CUNOP (operator, *error flag)

Errors detected: none

Calls: stack

CBINOP (operator, *error flag)

Errors detected: none

Calls: stack

GSEXPR (IL name of variable, *error flag)

Errors detected: variable uninitialized 17

variable inaccessable 18

Calls: CERRPT, symbol table, GEXPR

CONST (IL name of constant, type expected, *value, *error flag)

Errors detected: invalid length of constant 19

invalid character in an integer 20

invalid type of constant 21

character constant too long 22

invalid character(s) in boolean 23

Calls: FE, AMA, CERRPT, CONVRT, CNAMES (memory allocation)

```
**********************************************************
*                                                        *
*     CONVRT (*AM character, IL character value)         *
*                                                        *
* Errors detected: none                                  *
*                                                        *
* Calls: none                                            *
*                                                        *
********** THIS ROUTINE IS MACHINE-DEPENDANT !**********
```

GEXPR (token name, token type, flag if in register,
*result name, result type, flag if result in register,
*error flag)

Errors detected: binary operation expected, not found 42

Calls: stack, GCAND1, GCAND2, GCOR1, GCOR2, GUNOP, GARY,
GBINOP, CXTRCT, CERRPT

GCAND1 (token name, type, flag if in register, *error flag)

Errors detected: invalid type of operand 43

Calls: stack, AMA, register allocater

GCAND2 (token name, type, flag if in register, *result name,
result type, flag if result in register, *error flag)

Errors detected: invalid type 51

Calls: stack, AMA, register allocater

GCOR1 (token name, type, flag if in register, *error flag)

Errors detected: invalid type of operand 47

Calls: stack, AMA, register allocater

GCOR2 (token name, type, flag if in register, *result name, result type, flag if result in register, *error flag)

Errors detected: invalid type of operand 55

GUNOP (operation, token name, type, flag if in register, result name, result type, flag if result in register, *error flag)

Errors detected: invalid operation 10

invalid type of operand 11

Calls: AMA, register allocater, CERRPT

GBINOP (operation, token name, type, flag if in register, result name, result type, flag if result in register, *error flag)

Errors detected: invalid operation 4

exponentiation of non-integers 6

comparison of unalike types 7

arithmatic on non-integers 8

unknown operation 9

Calls: AMA, register allocater, CERRPT, memory allocter

GARY is described in the module 'Other Array Operations'

CXTRCT (array name, array type, flag if name in register,

index name, index type, flag if index in register,
result name, result type, flag if result in register,
*error flag)

Errors detected: array name in register 36
index not an integer 37
array name not in symbol table 38
array not accessable 39
array not initialized 40
name not name of an array 41

Calls: symbol table, register allocater, AMA, ARM, CERRPT

D.17 FORTRAN CODING PRACTICES IN THE CG

The CG's routines are coded in a way that I think will help others maintain them. The practices are as follow:

1. When an error is detected, a GOTO is made to a label number of 9000 or greater. Thus the in-line code is always concerned only the normal case.

2. An error causes a return code to be set to a unique number. The routine then returns. Thus there is no return to the in-line code.

3. Parameters are either input and unmodified or output and set by the called routine.

4. The following are the constructs used:

Construct		FORTRAN
if B then S1 else S2		IF (.NOT. B) GOTO L1
		S1
		GOTO L2
	L1	S2
	L2	
while B do S	L1	IF (.NOT. B) GOTO L2
		S
		GOTO L1
	L2	
case i of		GOTO (L1, L2, L3), I
1: S1	L1	S1
		GOTO LN
2: S2		
esac	L2	S2
	LN	

5. COMMON storge is assumed static. Named commons are used to hold special parameters in variables with mnemonic names.

BIBLIOGRAPHY

Aho, Alfred V. and Ullman, Jeffry D. Principles of Compiler Design Reading, Massachusetts: Addison-Wesley, 1977

Bartussek, Wolfram and Parnas, David L. "Using Traces to Write Abstract Specifications for Software Modules", UNC Report No. TR 77-012: University of North Carolina at Chapel Hill, December 1977

Bellovin, Stephen. Verifiably Correct Code Generation using Predicate Transformers Chapel Hill, North Carolina: University of North Carolina at Chapel Hill, in preparation

Brooks, Frederick P. Jr. The Mythical Man-Month Menlo Park, California: Addison-Wesley, 1975

Calingaert, Peter. Assemblers, Compilers and Program Translation Potomac, Maryland: Computer Science Press, 1979

Dijkstra, Edsger W. A Discipline of Programming Englewood Cliffs, N.J.: Prentice-Hall, 1976

Freund, Karl M. The design and Abstract Specification of a Translator Module Chapel Hill, N.C.: University of North Carolina at Chapel Hill,1979

George, James D. Jr. An Abstract Machine as an Aid to Compiler Portability Chapel Hill, N.C.: University of North Carolina at Chapel Hill,1979

Parnas, David L. "On the Criteria to be used on Decomposing Systems into Modules": Comm. ACM pp330-336, December 1972

Parnas, David L. A Program Holder Module Pittsburgh, Pa.:Carnegie-Mellon University Department of Computer Science, June 1973

Parnas, David L. "On the Need for Fewer Restrictions in Changing Compile-Time Environments" (co-author with W.D. Elliot and J.E. Shore), Proc. of the International Computing Symposium 1975, North Holland Publishing Co.

Strong, J. et al. "The Problem of Programming Communications with Changing Machines": Comm. ACM pp12-18, volume 1, number 8 1958