A C COMPILER FOR A CONTROL MICROPROCESSOR:

A CASE STUDY OF THE TMS 7000

by

RUSSELL BIESELE, B.S. in Eng. Physics

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

Accepted

December, 1986

ACKNOWLEDGEMENTS

I would like to thank Texas Instruments Incorporated for their funding of the initial phases of the project. I would also like to thank my wife, Judy, for her help in proofreading the manuscript. Finally, I wish to thank my committee: Dr. Hardwick for his efforts in correcting the thesis, Dr. Gustafson for his help in organizing things, and Dr. Archer for his frank and helpful advice.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1. INTRODUCTION 1

2. THE TMS7000 PROCESSOR 5

Technical Overview 5

Problems with the TMS7000 Instruction Set 13

3. THE C PROGRAMMING LANGUAGE 21

4. GENERAL DESIGN 31

Considerations 31

General Design Guideline Summary 38

5. IMPLEMENTATION AND DESIGN SPECIFICS 40

The Principle of Ongoing Design . 40

Understanding the Advantages of Assembly Language Coding 42

The Solution to the Problem 44

Register Access Method 47

Arithmetic Conversion Rules Changes 48

Implementation of the Symbol Table 50

Implementation of Storage Management 61

Design of the C Lexical Analyzer 73

The Implementation of Structures and Unions 82

Implementation of Symbol Indirection 103

Expression Analyzer: Design and Implementation 122

Overview 122

Unary Star Conversion 125

Tree Linearization 129

Constant Folding 134

6. PROBLEMS WITH THE IMPLEMENTATION AND DESIGN 142

Overview 142

Register Access Method 144

7. CONCLUSION 148

REFERENCES 151

LIST OF TABLES

1. Representative Instruction Formats for the TMS7000 8

2. Comparison of Port Usage in Single Chip and Microprocessor Mode 13

3. C Indirection Types 106

4. Indirection Stack Propagation Rules 109

5. Parser Flags Used in Indirection Translation 118

LIST OF FIGURES

1. TMS7000 CPU Registers 6

2. Comparison of Single Chip Mode and Microprocessor Mode Configuration 11

3. TMS7000 Memory Map 14

4. Accessing Hardware Dependent Addresses . 24

5. Conditional Value Assignment 25

6. Precision Expansion 26

7. Questionable Use of Pointer 28

8. Questionable Use of Pointer Clarified .. 29

9. Recursive Nature of C Declarations 36

10. Recursion Induced Difficulties in Parsing C Declarations 37

11. TMS7000 Register Access Method 48

12. Register Access Simplified 49

13. Symbol Table Organization 52

14. Symbol Table Scoping List 53

15. Consistent Function Declaration, Definition, and Reference 59

16. Inconsistent Function Definition and Reference 60

17. Storage Management Scheme 64

18. Illustration of Dead Stack Zone 70

19. Illustration of Stack Margin Area 72

20. Organization of the Compiler's First Half 74

21. Comparison of Unambiguous and Ambiguous Syntax 76

22. Context Induced Difficulties 81

23. Definition of Structure Terminology 84

24. Nested Structure Declarations 86

25. Member Offset Calculation Test Program . 88

26. Detection of Endlessly Recursive Structure Declarations 91

27. Structure of a Record Analysis Tree 96

28. Member List Example 102

29. Indirection Stack Example 107

30. Semantically Restricted Operations 111

31. Function Pointer's Asymmetry 112

32. Scalar Separation of Array References .. 114

33. Implementation of Size Stack 116

34. Definition of Indirection Region 120

35. Assignment Context Determination 127

36. Tree Linearization Transformation 131

37. Group 1 Transformations 137

38. Group 2 Transformations 139

39. Group 3 Transformation 141

40. Register/Memory Access Special Case 146

CHAPTER 1

INTRODUCTION

The TMS7000 C compiler project was begun by Texas Instruments Inc. in an effort to make their microprocessor more competitive with other manufacturers' single chip microprocessors. The compiler was to be one of the first "full feature" high level language compilers produced for a single chip microprocessor, and the first compiler to provide intimate access to microprocessor dependent hardware in multiple chip applications.

One previous effort, the Small C Compiler, implemented a highly restricted subset of the C language on an 8080 microprocessor
[11]. This subset was tailored to allow both the compiler and its application programs to run in the 8080's 64K address space limit. The data types were limited to 16 bit integer and 8 bit character to match the processor's limited precision. Pointers were only allowed to point to one of the above data types or a function.

The limitations stated above allowed progress to be made towards a C language for small and single chip

microprocessor systems because the resulting language was more attuned to the needs of the programmers of these systems. The most important of these needs was the ability to allow the restrictions of the microprocessor's instruction set to be imposed upon the source code of the program. This allowed the programmer to minimize the number of "simulated operations" by a conscious alteration of his algorithm.

However, the two register model used by Small C is too restrictive for most applications. Even single chip microprocessor applications need to use a full set of registers. Structures, which are not implemented by Small C, are the most efficient method for storing the multiple type tabular data often required by device service routines. Common microprocessor hardware features such as interrupts and I/O ports were not supported at all.

Another previous effort, the PL/M compiler produced by Intel Corporation[13], made further progress towards the support of microprocessor hardware. This implementation of the PL/I language supported the interrupts and segmentation of the 8086 family of microprocessors. However, because it is based on PL/I, PL/M does not give the programmer good tools for accessing a microprocessor at the machine code level.

The TMS7000 C Compiler blazes a new trail because it attempts to provide the advantages of Small C and PL/M without sacrificing the major features of a "full C" implementation. It also attempts to handle in a general way the widely varying hardware implementations unique to single chip and control microprocessor applications. The goal of its design is to provide superior hardware access in a constrained environment without sacrificing any of the features or generality of the C language. To successfully achieve this goal the following list of requirements had to be met by the code generated by the compiler.

1. It had to be compact.
2. It had to suitable for ROM.
3. It had to be time efficient. Nonexistent operations can't be simulated by subroutines.
4. It had to provide low level access to microprocessor hardware directly from C source code. Restricting hardware access to calls of machine code library functions is not acceptable.
5. It had to be able to operate without a stack when necessary.
6. It had to operate on systems whose memory is fragmented.

6. It had to provide constructive cooperation between interrupts and C subroutines. This means that interrupts must be defined and controlled from C.

The TMS7000 hardware underlying the above requirements is discussed in Chapter 2. Chapter 3 contains a summary of existing facilities in the current C language which are relevant to microprocessors. Chapter 4 gives a general overview of the compiler design by matching compiler requirements to general approaches satisfying those requirements. Chapter 5 describes some important parts of the compiler implementation in detail. Chapter 6 discusses the problems with the implementation. Chapter 7 concludes the thesis by summarizing what was gained from the project.

CHAPTER 2

THE TMS7000 PROCESSOR

Technical Overview

The TMS7000 is an 8 bit microprocessor designed to be used in process control and device control applications. Current models contain 256 8 bit registers, a mask programmable ROM, a serial communications port, 3 event timers, and up to 256 external 8 bit ports. It has an instruction set which can be divided into core and noncore instructions. The noncore instructions may be replaced by user defined instructions at the time of manufacture. This flexibility allows the mass production user to tailor the microprocessor to their needs. The customizing is made possible by its microcoded architecture[2].

The instruction set of the TMS7000 is designed to be efficient, minimal, and to allow the microprocessor chip to function as a single chip microcomputer. A 256 byte RAM is integrated into the CPU hardware so it can be used in single chip applications. This RAM has an absolute address of 0-FF hex and can be addressed either via special 8 bit addressing modes or via a 16 bit absolute address. Access via the 8 bit addressing mode is efficient enough to allow the RAM to be used as

256 8 bit registers. Figure 1 shows the layout of these registers.

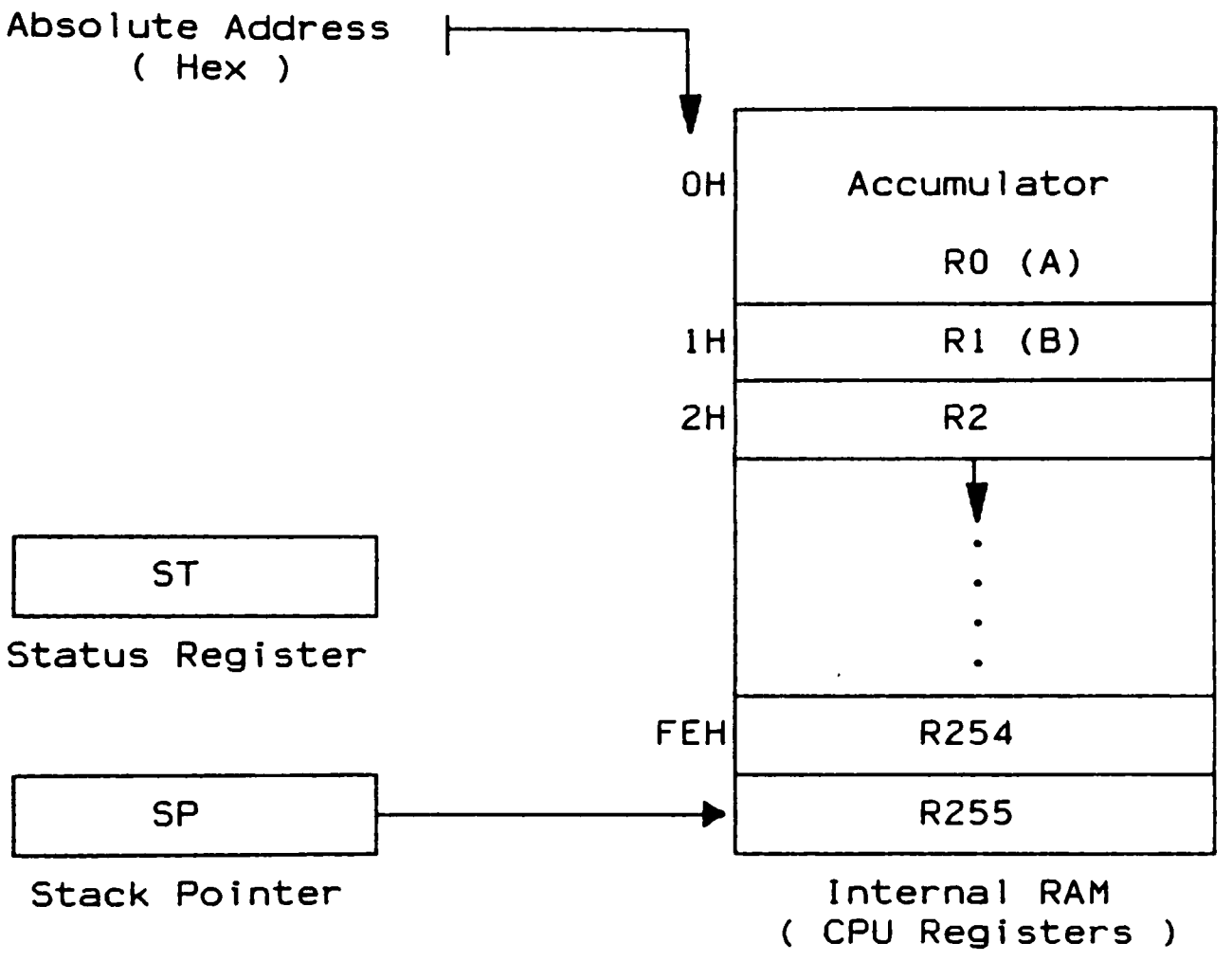


Figure 1: TMS7000 CPU Registers

The 8 bit address mode can be used in two ways. The first way is to specify an 8 bit constant address byte after the opcode of an instruction. This use of an 8 bit constant address exactly mimics the use of a

normal 16 bit constant address. Thus, one or two bytes can be saved on each data instruction. Space saving capability is uniquely important to a single chip microprocessor since generally a program must fit entirely into the limited on chip ROM space.

The second way the 8 bit address mode may be used is via indirect addressing using the stack pointer register. This register, as shown in Figure 1 above, is not part of the memory address space like the other CPU registers. It may only be used as a stack pointer and thus only allows access to the internal RAM locations via push, pop, and call instructions. The programmer is forced to locate this stack in the internal RAM.

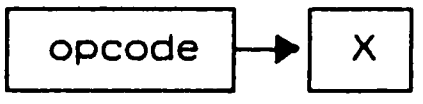
Some of the stack instructions use an address mode called implied addressing. Implied addressing instructions save code space by not requiring instruction bytes specifying the source and destination operands. Either one or both of the operands are chosen by the actual choice of the opcode used and may be either R0 (the A register), R1 (the B register), or the internal RAM location pointed to by the stack pointer.

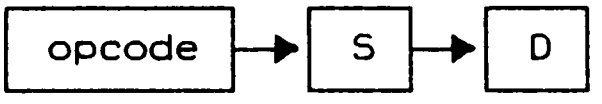
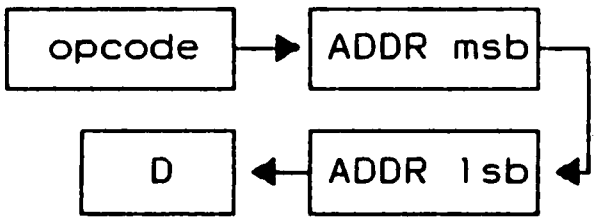
A representative sample of instructions is shown in Table 1.

Table 1: Representative Instruction Formats for the TMS7000

Instruction Type	Representative Instructions	Instruction Format

Implied Addressing	push A
	pop ST
	mov B,A
	mov A,B
	dec A
		

Single Register (8 Bit Addressing)	dec RX
	mov RX,A
	mov A,RX
	push RX
		

Dual Register (8 Bit Addressing)	mov RS,RD	
Extended Addressing (16 Bit Addressing)	movd %ADDR(B),RD	

The principle of accessing a limited section of memory with an 8 bit address constant is also used to create a set of 256 8 bit memory mapped ports for the TMS7000. These ports occupy the absolute address range 100 hex to 1FF hex. This range is called the

"Peripheral File." It contains both internal I/O and timer registers, and locations reserved for use with registers in external hardware of the user's choice. The number of locations is determined by whether the CPU is in the microprocessor mode or single chip mode.

In single chip mode, no locations are reserved for the user since there are no external data busses. Instead, the pins normally devoted to address/data bus signals are used as two input/output ports. Additional port locations normally available in microprocessor mode are reserved and used to access these additional internal ports (Ports C and D). The user's assembly code directly controls the transactions with any busses attached to port C and D. The remaining microprocessor mode port addresses are reserved in single chip mode for future TI hardware expansion.

In microprocessor mode, CPU ports C and D are used by the CPU to access an external address/data bus. The ports are therefore under direct microcode control and can't be accessed via user assembly instructions. External ports (external hardware registers) are accessed automatically at the command of the user's assembly code by the CPU's microcoded program. Thus, in this mode, access to external ports is transparent and these

ports take on equality with the remaining internal ports (internal hardware registers).

Figure 2 contains a comparison between the configurations of the TMS7000 for the microprocessor mode and the single chip mode. Table 2 shows a comparative memory map for the I/O ports in the two modes.

The remaining part of the TMS7000 address space above the I/O port addresses is allocated for user memory and on-chip ROM/EPROM. The entire memory map for the TMS7000 and its mode dependencies are shown in Figure 3.

In the single chip mode, all addresses between the end of the Peripheral File and the on-chip ROM/EPROM are unusable. In microprocessor mode, there are two possible memory maps. When the MC pin is held low, then on-chip ROM/EPROM is enabled and only addresses outside the ROM/EPROM address range are passed to external memory. When the MC pin is held high, the on-chip ROM/EPROM is disabled and all addresses above the Peripheral file are passed to the external memory. The MC pin true mode is primarily used on models of the TMS7000 which contain no on-chip ROM/EPROM. This model is commonly used in developmental work.

It should be noted that the TMS7000 actually has 4 modes of operation. The modes detailed were at each end of this mode range. The reader is encouraged to consult the references for further details. This completes the discussion of the TMS7000 hardware. The next section details some of the problems associated with the TMS7000 architecture and instruction set.

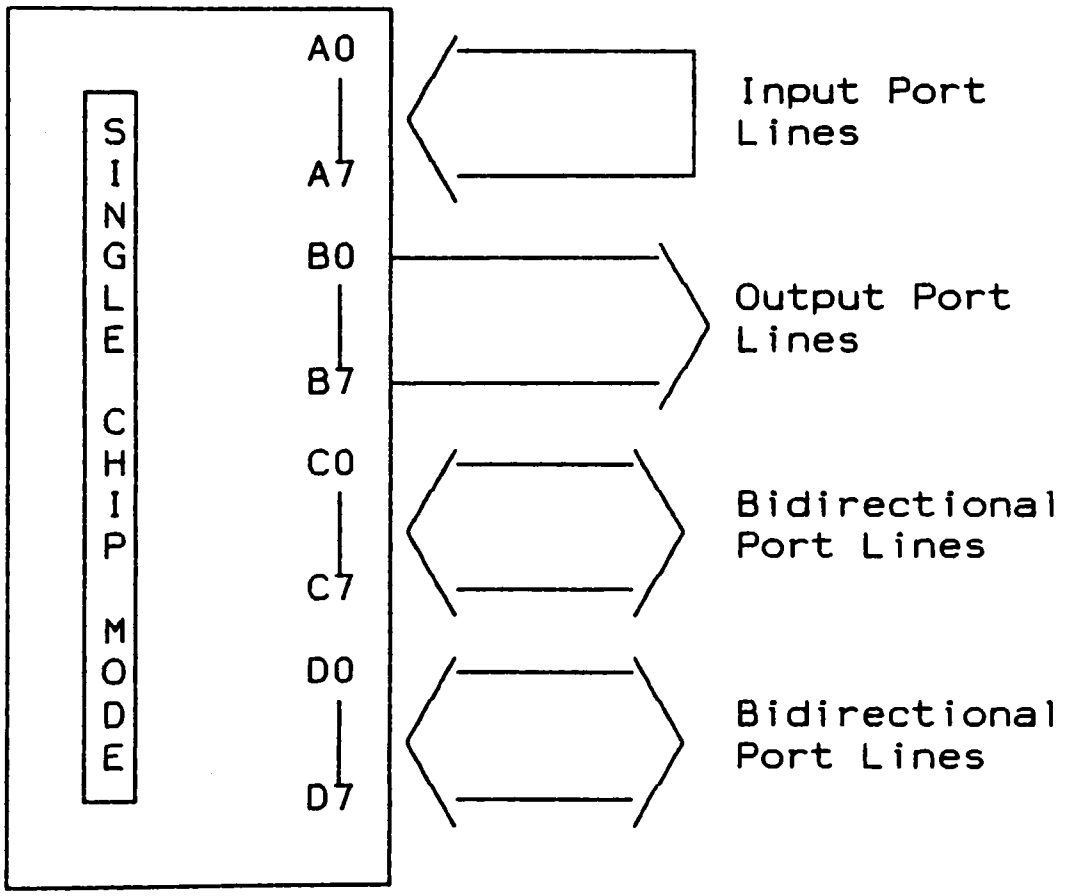


Figure 2: Comparison of Single Chip Mode and Microprocessor Mode Configuration
Part 1

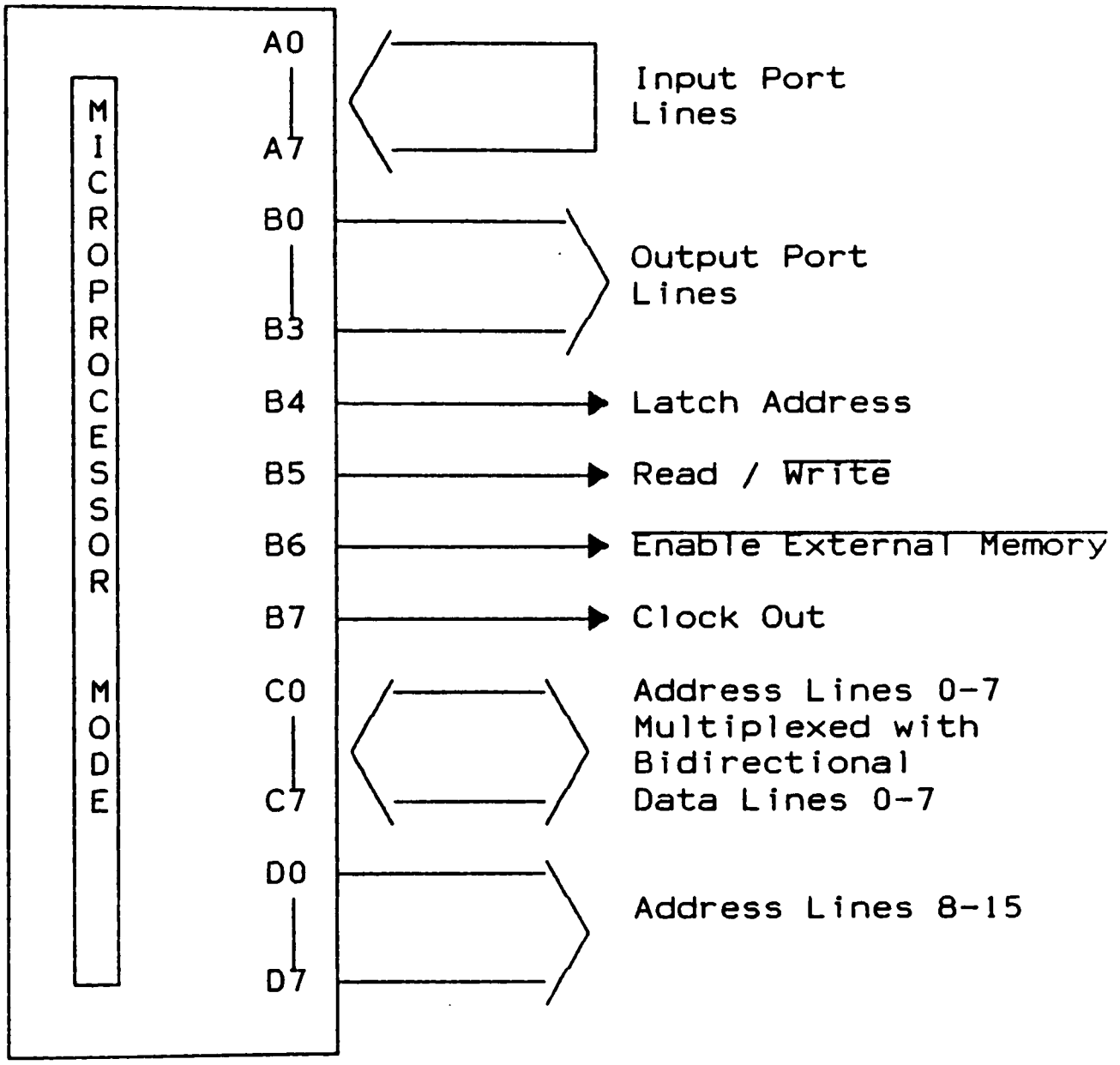


Figure 2: Comparison of Single Chip Mode and Microprocessor Mode Configuration
Part 2

Table 2: Comparison of Port Usage in Single Chip and Microprocessor Mode

Port #	Hex Addr.	Single Chip Usage	Microprocessor Usage
0	100	I/O Control	I/O Control
1	101	Reserved	Reserved
2	102	Timer Data	Timer Data
3	103	Timer Control	Timer Control
4	104	Port A Data Value	Port A Data Value
5	105	Reserved	Reserved
6	106	Port B Data Value	Port B Data Value
7	107	Reserved	Reserved
8	108	Port C Data Value	User Defined
9	109	Port C Direction	User Defined
10	10A	Port D Data Value	User Defined
11	10B	Port D Direction	User Defined
12-255	10C-1FF	Reserved	User Defined

Problems with the TMS7000 Instruction Set

Although the instruction set and architecture of the TMS7000 serves well enough for small, single chip applications, microprocessor mode applications are not well served. This deficiency is important because most of the larger more difficult control applications use

microprocessor mode. One of the first problems encountered is the design of the processor's stack. Every time a push is made to this stack, a register is lost.

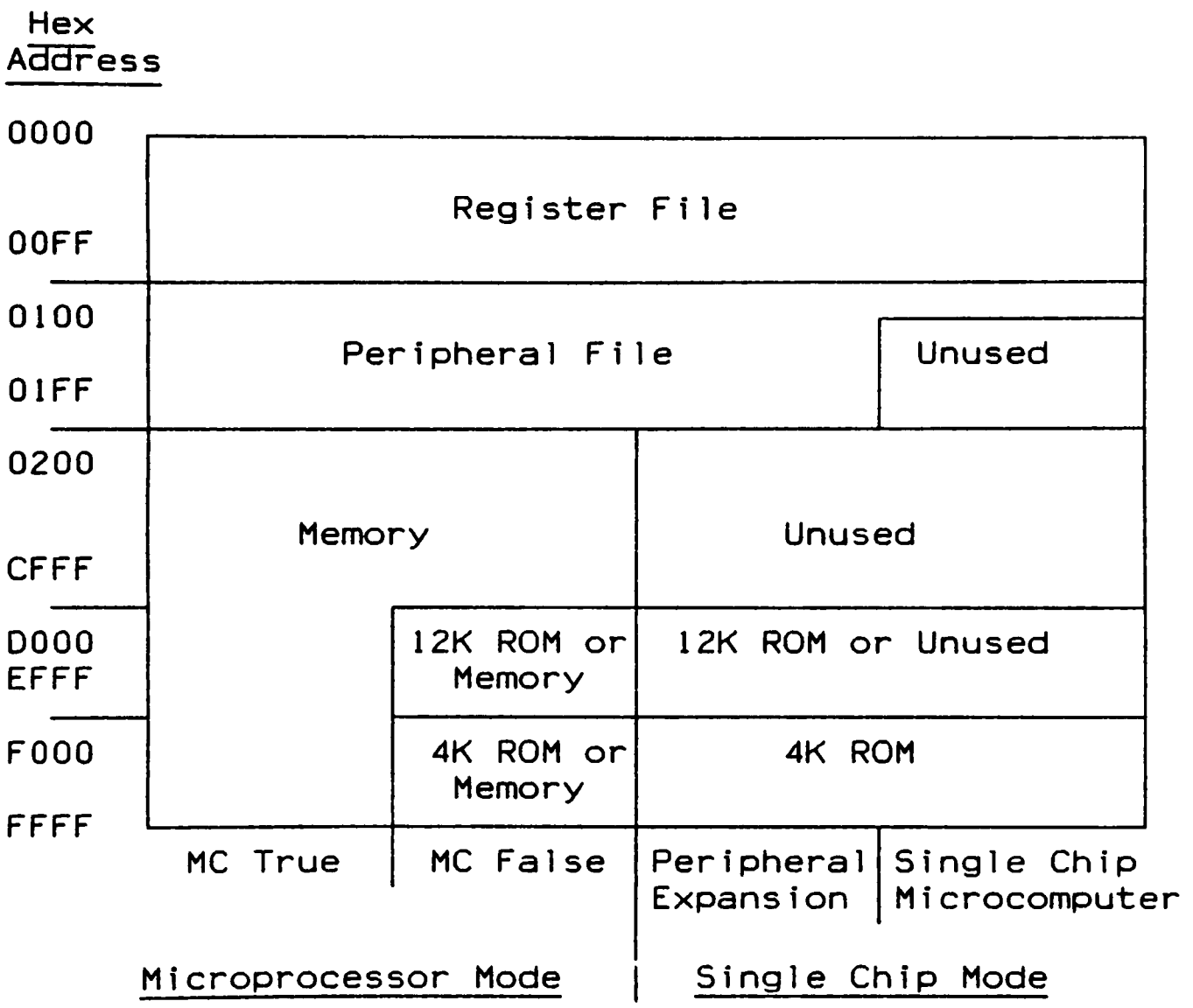


Figure 3: TMS7000 Memory Map

Although one can implement a software stack in external memory using the processor's indirect addressing modes, the subroutine call, the push, and the pop instructions

will only use the internal hardware stack[1]. Thus, the subroutine call/return mechanism forces the user to set up and maintain the internal stack even if it is undesirable.

This restriction was not viewed in the processor's original design as a disadvantage because the processor's primary use was as a single chip computer. Address and data bus pins were not required on its standard 40 pin package since the microprocessor was self contained. The extra pins on the microprocessor chip were available as single bit input or output I/O ports. Thus, the microprocessor could be used as a disk controller with each bit either sensing or controlling the disk drive mechanism. Or, by using the internal timers, the processor could measure the pulse width of an incoming signal (the time an input bit remained at logic 1) and send out whatever response was appropriate[2]. By simple assembly programming, the equivalent of a complex hardware logic circuit could be produced.

This view changed, though, as technology progressed and it became desirable to produce "intelligent devices." These devices require the larger program and data memory possessed by a normal microcomputer, but the required compact size or low chip count demanded a solution similar to a single chip system. Such an

application can occur in a speech synthesis system[14], for instance, where large pools of data are required and the chip count must be kept low for cost effectiveness. In the particular system outlined in the reference, the CPU operated in microprocessor mode to load up a speech pattern and a small program into the internal RAM from an external EPROM and then switched to single chip mode during speech generation. This switch allowed the microprocessor to simulate some of the external chips that would have normally been required to interface the microprocessor with the speech chip. This simulation was performed by allowing the software to directly control the data bus. Once speech generation had been completed, the software switched the microprocessor back into microprocessor mode and allowed the microprocessor microcode to gain access to the external data bus again.

There are problems with using the TMS7000 in microprocessor mode. First of all, the addressing modes of the instruction set are very limited. This limitation is a natural outcome of a single chip environment where no or very little external memory is expected to be present. Secondly, the instruction set is not symmetric and has instructions missing. For example, while there is a DECD instruction to do a double

precision (16 bit) decrement of a "register pair," there is no INCD instruction to do a double precision increment[1]. This missing instruction is important because one may use a "register pair" to address an external memory location. If the previous location is desired the DECD (decrement double) instruction may be used to decrement the memory address in the register pair. If the next memory location is desired, the programmer is out of luck.

In defense of the TMS7000 instruction set, it has one of the most powerful instruction sets for I/O port control and strictly 8 bit register to register operations. This again is in accord with the single chip philosophy. The TMS7000's role is that of an external device controller. Therefore, these instructions could not be sacrificed for better addressing modes even in the microprocessor mode of operation. However, because an intelligent device is clearly more complex than a single chip computer, assembly programming becomes an ordeal at best. A perfect example of this complexity is a microprocessor controlling and monitoring various functions of an automobile. Engine monitoring is not much of a problem, but when this is combined with voice synthesizer enunciations of engine malfunctions, voice reminders to the driver of overdue maintenance, and

cross country navigational systems, the programming becomes complex. These features and more are becoming standard on many of the newer cars.

With this increasing tendency toward better human interface and more complex control programming, the power of a high level language becomes required. But to be efficient, most high level languages require either the host processor to have flexible and numerous addressing modes or to have a reduced instruction set computer architecture (RISC)[4].

The TMS7000 clearly lacks addressing modes which processors outside the RISC category should have. In particular, the Motorola MC6801, which is comparable to the TMS7000, has an addressing mode which uses the sum of a 16 bit index register and an 8 bit offset constant byte to form a memory address[3]. Stack addressing is facilitated by allowing values in the 16 bit stack pointer and 16 bit index register to be interchanged[15]. Thus the stack may be located either in its internal or external RAM and stack frames of less than 256 bytes can be efficiently accessed.

The TMS7000 only has an index addressing mode which uses the sum of a 16 bit address constant and an 8 bit index register. This type of indexing is useless

for stack indexing unless a stack with a maximum size of 256 bytes and fixed memory position is acceptable.

In some ways, the TMS7000 is comparable in simplicity to a RISC processor. A RISC processor particularly suited to high level languages is the one implemented at University of California, Berkeley[4]. It has a large number of registers which allow a separate set of registers to be used for each subroutine. In that implementation, the registers are "windowed." This means that although the registers are known by the same set of names in every subroutine (R10 - R31 for instance), these registers refer to different storage locations within the processor. The only exceptions to this scheme in the Berkeley RISC processor are the "global registers," which always refer to the same storage location within the processor.

The TMS7000 has no provisions for windowing registers. All of its registers are global. In order for private registers for subroutines to be implemented, the compiler would have to know the complete history for the program being compiled. This would be difficult for a program with multiple source files. Further, part of the RISC architecture's speed is due to its simple instruction set being implemented as combinational logic instead of microcoded logic. The TMS7000 instruction

set is microcoded[2]. Thus, the TMS7000 does not fit into the RISC category and design of the code sequences generated by the C compiler will require careful analysis of the number of microcode cycles required for each proposed instruction path.

The applications that the TMS7000 is commonly used in demand that the I/O ports be accessible[5]. High level languages tend to be designed to be machine independent and therefore intentionally do not normally define entities which allow access to something as machine dependent as I/O ports. This restriction as well as those imposed by the instruction set will be addressed later when the design goals for the TMS7000 C Compiler are stated.

CHAPTER 3

THE C PROGRAMMING LANGUAGE

The C programming language has grown from an obscure systems programming language used within the Bell Laboratories into a language that is used widely in the computer industry[6].

Programming on microprocessor based systems requires that the programmer become intimately involved with the system's hardware. Most former high level languages were excluded from use with these systems because they did not provide direct access to hardware within the microprocessor chip or within the microprocessor system. The primary language thus became whatever native assembly language was available for the microprocessor. This meant that the microprocessor independent sections of the program had to be written from scratch for every new system. Further, assembly language blurred the distinction between machine independent and machine dependent code. Even worse, the boundaries of application independent blocks of code (software tools) were blurred and their modularity severely compromised. This meant that the tools could not be recognized or extracted successfully.

With the advent of the C language, this picture has radically changed. C was able to replace assembly language in most microprocessor applications. The innovations of C driving this change were as follows:

1. Access to memory mapped I/O and CPU hardware became possible via addresses "calculated" by the programmer. Thus, memory access was only limited by the address space of the microprocessor. Most other high level languages limited memory access to declared objects. Since much of microprocessor programming is directed at control of intrinsic or externally attached hardware, this feature immediately thrust C into use.
2. Memory access was via flexible addressing modes implemented in a manner precisely mimicking commonly used microprocessor addressing modes. Most important of these modes was an infinitely extensible chain of indirect addressing. The precision of each memory access was controlled by pointer typing, and these types corresponded to intrinsic hardware data types.
3. Operators allowing bit manipulations common to most microprocessors are provided. The C

language specifies these operators in a way which allows them to be implemented by a single microprocessor instruction in most cases. Operators which fall into this category are as follows:

A. Left Shift.

B. Right Shift.

C. Increment.

D. Decrement.

E. Bitwise Or.

F. Bitwise And.

G. Bitwise Exclusive-or.

H. One's Complement.

Because of the above innovations, the C language allows intimate access to a processor's memory mapped hardware in a processor independent manner[6,9]. The compact symbolic notation of these operators afforded a much clearer representation of the programmer's algorithm than assembly language. Providing the algorithm makes sense for other machines, the program can be ported to another machine by simply recompiling it on the target machine. And most importantly, because these C operations translate almost directly into the target machine's assembly code, there is very little penalty in efficiency for using C. The integration of the above

three points into a common microprocessor code fragment is shown in Figure 4. The code in the figure sets the least significant bit of a 16 bit hardware register located at address 1FF hex.

```
int *ip ;

ip =  (int *)  0x1ff ;        /*Statement #1*/
*ip = *ip | 01 ;              /*Statement #2*/
```

Figure 4: Accessing Hardware Dependent Addresses

Modularity and readability of the program are improved by the use of the program flow control structures found in C. C contains all of the flow control structures necessary to implement fully structured programming[7]. Also, the set of logical operators contained in C allow some assembly language programs with complex flow patterns to be represented in a compact symbolic notation. For instance, a common assembly language construct is to assign one value to a memory location if a logical expression is true and a different value if the logical expression is false. This simple task would be coded in assembly language using a compare instruction and a conditional jump. When this

block of assembly code is embedded within code containing other conditional jumps, the code becomes difficult to read. However, this assembly code block can be directly translated into C with no loss of efficiency and a large gain in readability as shown by the example in Figure 5.

```
memory_value = logical_expression  ?
               true_value : false_value ;
```

Figure 5: Conditional Value Assignment

In addition to clarifying the code with concise operators, C, like many other high level languages, provides a variable typing and a type checking mechanism. Typing helps prevent some of the more common programming errors. However, in some languages, typing can restrict the programmer to the point that he is not able to express many low level algorithms efficiently. C, unlike other high level languages, provides an orderly escape mechanism which preserves error detection capabilities while allowing the flexibility that low level programming demands. An example of this feature can be found in precision truncation and precision

expansion. The coding example in Figure 6 illustrates precision expansion:

```
char cd = '0' ;
int i = 5 ;
int isum ;

isum = i + cd ;
```

Figure 6: Precision Expansion

In this example, an 8 bit character variable cd is being added to a 16 bit integer variable i and the sum is being stored in the 16 bit variable isum. C's typing mechanism helps the programmer by expanding the precision of the value of the variable cd to 16 bits prior to the addition. This expansion would have to have been performed by the programmer anyway if he had coded this addition in assembly language. Thus, the typing mechanism performed a service to the programmer and relieved him of this drudgery. By the way, isum would as a result of the addition contain the character code for 5, thus converting the binary 5 to a character 5. If isum is now assigned to cd by the statement cd = isum ; , the precision of the value from isum is automatically truncated to the 8 least significant

bits. The value is then stored in cd. This is desirable because it preserves the character code in the lower 8 bits. C always truncates by taking the least significant portion of a value since this is generally more useful to the programmer. In both precision truncation and precision expansion, C produces no error messages and does not inhibit the programmer in any way.

C's typing mechanism mainly restricts programmers only where serious mistakes are likely. One heavily scrutinized area is assignment via address pointers. A C address pointer typically allows a program to arbitrarily write practically anywhere in addressable memory. Although the compiler can't know if a pointer contains a valid address, it can flag questionable practices and ask the programmer if he is sure his code is correct. An example of a questionable coding practice which would be flagged is shown in Figure 7.

In Figure 7, ip is a pointer to a 16 bit integer location and cd is an 8 bit character location. The statement labeled #1 assigns the address of the character variable cd to the integer pointer ip. Depending upon the compiler implementation, this statement would be flagged with either a warning or an error stating that the type of the pointer does not match the type of the location which the address identifies.

```
char cd ;
int *ip ;

ip = &cd ;    /*Statement #1 */
*ip = 5 ;    /*Statement #2 */
```

Figure 7: Questionable Use of Pointer

C chooses to flag this statement because it is questionable that a programmer would really intend to do this. The reason it is questionable is illustrated by statement #2. In this statement, 5 is a 16 bit integer. The lower 8 bits of 5 contain 5 and they are assigned to the character variable, cd. However, the upper 8 bits of 5 are zero, and they are assigned to the lower 8 bits of whatever variable happens to adjacent to cd. Thus, the adjacent variable is overwritten. If the programmer really intends this overwrite to happen, he is not prevented from doing it. He is just being asked to clarify his intentions. An example of this clarification is shown in Figure 8.

Statement #1 has now been clarified by the addition of (int *), which is called a cast. A cast is a unary operator which allows the programmer to set the type of an object to any legal type of his choosing.

The cast operator in statement #1 tells the compiler that the type mismatch is intentional. Thus, one reason C is becoming popular for use with microprocessors is because of its sensible yet escapable typing mechanism.

```
char cd ;
int *ip ;

ip =  (int *) &cd ;    /*Statement #1 */
*ip = 5 ;              /*Statement #2 */
```

Figure 8: Questionable Use of Pointer Clarified

In short, all the features of C can be summarized in one phrase: freedom of expression for the programmer. C spans a large dynamic range of programming levels, from assembly level operations to those equivalent to Pascal. It has the directness of Fortran as well as the data structures of PL/I. This expanse of dynamic range and flexibility makes it plausible for C to become the universal microprocessor language. The low level access afforded in C makes it possible for the TMS7000 and other control microprocessors to have their first high level language. The high level features and compact symbolic notation of its low level operations

make C a language highly demanded by microprocessor users.

The list below summarizes the features of C which have been critical to its acceptance as the microprocessor programming language.

1. Direct access to memory mapped hardware.
2. Indirect addressing modes similar to those used by a microprocessor's instruction set.
3. Bitwise instructions directly corresponding to common microprocessor bitwise instructions.
4. High level flow control structures based on low level microprocessor operations.
5. Escapable typing mechanism.

CHAPTER 4

GENERAL DESIGN

Considerations

The first task in designing a piece of software as large and as complex as a C compiler is to specify a general approach and philosophical guide which can be followed during the detailed design and implementation phase. The initial facts governing the composition of this guide for the TMS7000 were as follows:

1. The end users of the compiler will be entrenched assembly language programmers who have a low opinion of high level languages. Very few, if any, higher level languages have had the code efficiency and hardware accessibility control assembly programmers require.
2. Due to their complexity, compilation of C declarations will be one of the most difficult tasks to accomplish.
3. Optimizations involving code motion and gross modifications of the code are unacceptable since the primary use of the compiler will be that of an assembly code generator. The logical flow of the program needs to be preserved.

4. With code motion not allowed, only data flow optimizations and optimization of instruction selection remain. These two methods require detailed information from the following major areas:

   A. Storage type. The compiler's code generator must know if a variable is automatic (stack storage, stack pointer relative address) or static (memory address space, absolute address). In the case of the TMS7000, this information allows the following optimizations:

      a.) Recognition of special addresses or entities. For example, if a data entity represents a hardware register such as a CPU register or an I/O port, then the appropriate special purpose instruction can be generated.

      b.) Reduction of access complexity. Access to some data entities may require complex pointer calculations or long instruction sequences due to poor addressing modes for the entity's storage type. If the

context of an entity's usage is known, the data value or address of the entity may be cached in a register. The context of the entity's usage also allows its retention time in a register to be based on a prioritized scheme. The entities with the highest degree of access complexity and highest frequency of access are given the highest priority.

B. Context of a data entity's storage precision in an expression. This information allows the most efficient instruction sequence to "grip" the data. The context information allows the statement "goal" to be discovered. The following example shows the importance of context in selecting the proper grip: An 8 bit CPU (such as the TMS7000) wishes to perform C = A + B, where C and A are 8 bits in precision and B is 16 bits in precision. The possible grips from worst to best are as follows:

a.) Standard C Grip: No context information is available. The only safe way to perform the calculation is to do it in the same precision as the operand with the largest precision. C then implicitly truncates the result to 8 bits prior to storing it in variable C.

b.) Context Optimized Grip: The "goal" of generating an 8 bit result is realized. Precision above 8 bits between the initial values stored in memory and the goal is useless. An 8 bit grip is used. No precision expansion or extended precision calculations need to be performed.

5. Error detection and reporting will have to be improved over previous C compiler efforts. Specific, unambiguous error messages will aid the application programmers in learning C. Methods in reducing error message cascading will reduce their confusion.

6. The early C implementations were top-down[12]. The grammatical design of C favors a top-down approach[10].

7. At the time of design, no documented YACC or similar parser generator was commonly available for the development machine (IBM PC Compatible).

Because of the above considerations, a top-down approach was chosen. The nature and complexity of C declarations was a major deciding factor. Although declaration parsing can be implemented using a bottom up scheme, extra global variables and a stack external to the bottom up processing is required[10]. In a sense, the bottom up parser would be simulating a top-down parser. This top-down parsing of C declarations is forced by their recursive nature, which is painfully illustrated in Figure 9.

Declaration #1 in Figure 9 is declaring memspace to be a pointer to a character array. The character array contains the same number of elements as the size in bytes of the declared record, test. Declaration #2 in Figure 9 is declaring fp to be a pointer to a function which returns a pointer to an array of 20 integers.

The identifier being declared or the "focus" of the declaration appears at the lowest level in a declaration. The operators that appear in this lowest level or focusing region determine the size of the storage

reserved. The inner most set of parenthesis containing a unary star operator define the bounds of the focusing region. Figure 10 shows a declaration decomposed into its component parts.

```
char (*memspace)[sizeof(struct test {
                              int i1 ;
                              char c ;
                              int i2 ;
                              }
                         )
                  ] ;       /*Declaration #1*/

int (*((*fp)()))[20] ;  /*Declaration #2*/
```

Figure 9: Recursive Nature of C Declarations

Outside the focusing region, each parenthetic level with one or more unary stars directly adjacent to its left parenthesis defines an indirection region. When a declaration is parsed, the focusing region must be decoded first. Then each indirection region from the inner most one outward must be decoded. During the decoding of the right side of a region, the semantic restriction that () and [] may not appear in the same region must be enforced.

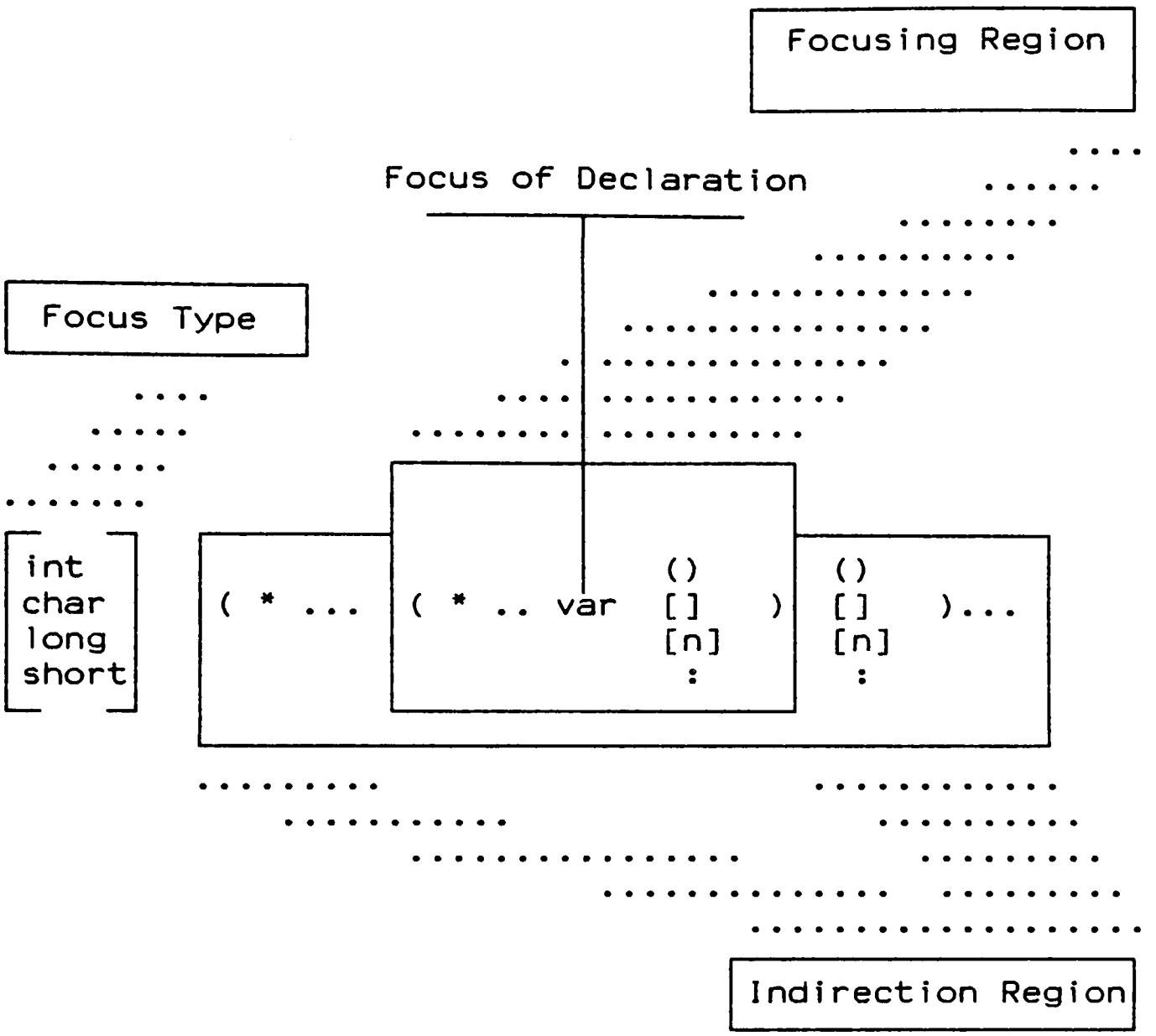


Figure 10: Recursion Induced Difficulties in Parsing C Declarations

In a bottom up parser, it is possible but difficult to impose this restriction. The main obstacle is that a region is not recognized until the right parenthesis corresponding to the region's left hand

parenthesis is seen (not all right hand parentheses are region boundaries). The () and [] are seen prior to region recognition. A top-down recursive descent parser breaks down the above problem into modular independent units. This breakdown mirrors more closely the human conceptual analysis. It is possible to see a direct cause and effect between parsing problems and the parser code itself. This flexibility is important in an "experimental" compiler.

Top-down parsing also aids the compiler in producing more specific and informative error messages. At any point in a statement, the top-down parser knows what tokens could legally occur next. By anticipating what tokens would be next if a common coding mistake occurred, a specific error message can be generated as opposed to a vague generic one.

General Design Guideline Summary

The following statements summarize the guidelines that can be drawn from the above discussion:

1. Top-down parsing will be used.
2. Context information will be passed down from the parser to the code generator for improved but assembly programmer compatible optimization.

3. The context knowledge a top-down parser possesses will be combined with a knowledge of common coding mistakes to produce better error messages.
4. Ways will be found to increase hardware accessibility from C.

CHAPTER 5

IMPLEMENTATION AND DESIGN SPECIFICS

In this chapter the design and implementation of the compiler will be discussed. The first 5 sections outline some preimplementation design specifics which were demanded by the TMS7000 processor. The next 6 sections will discuss the ongoing design during the implementation and describe the algorithms that were discovered during that process. The next chapter will discuss the problems encountered in the first prototype during its implementation.

The Principle of Ongoing Design

The TMS7000 compiler was envisioned from start to finish as a software engineering project instead of an exercise in compiler theory. The difference between theory and engineering is that theory tends to be developed under ideal conditions. Engineering has to deal with the realities of the machines in use and the demand of the human users of the compiler. Unlike a theoretical exercise, successful development of a working compiler does not mean the project is a success. A successful project is attained when a working compiler meets performance expectations and works in the target environment.

This does not mean that compiler theory was abandoned but rather the numerous theoretical sources became a source of ideas. The emphasis here is that the ideas were more important than the mechanics of the theory. Once the correct mix of ideas to follow could be selected, then all detailed design and source code production could follow.

One of the major problems in selecting these ideas is that C is a language that is not well specified. This is especially evident when programs with coding styles outside the mainstream are tried. Another problem is that most compiler work available to the nontheoretician deals with context-free languages. C is not a context free language. Traditional approaches such as the one taken with the Portable C Compiler[10] tend to treat C as a context free language and then deal "on the side" with the aspects violating this context free nature. In other words, the compiler is "engineered" until it works. The main problem facing the new compiler writer is that much of this engineering information is not published and remains a "trade secret" of the proprietary company.

The net result is that experimentation is required to obtain this "engineering" information. The experimentation approach was used in the production of the

TMS7000 C Compiler. This experimentation consisted of either coding an algorithm in C or stepping through it on paper in a thought experiment fashion. In either case, a successful result was not definitive. Quite often, earlier portions of the compiler were later discovered to have flaws and a engineering revision of the earlier portion occurred in parallel with the newer sections being designed.

Throughout the discussion of the compiler in this chapter the phrase "first prototype" is used. The first prototype was never a completed compiler but rather a version in which the overall design remained relatively stable. At some point during the project, several flaws were detected in the design and a major revision in the overall compiler design occurred. Due to time limitations though, this revision never led to a completed compiler. It is presented as a point of comparison to demonstrate the ongoing design required to produce the TMS7000 C Compiler.

Understanding the Advantages of Assembly Language Coding

In order to be acceptable, the C compiler for the TMS7000 will have to be capable of performing tasks nearly as efficiently and with at least as much programming ease as assembly code. The features offered in

assembly language coding that were formerly not offered in higher level languages must be identified. The following features were identified in TMS7000 assembly language programs:

1. Most arithmetic and data manipulative functions are performed using CPU registers only. Use of memory fetches and stores cause a rapid increase in code size and a resulting decrease in execution speed. This is a result of the poor addressing modes.
2. The assembly language programmer, in more critical single-chip applications, will write code which allocates the use of certain registers to each routine. Parameters to the routine are passed via specified registers.
3. Arithmetic operations are performed using only the required precision (required number of registers).
4. Time and space efficient instructions unique to the TMS7000 instruction set are coded. Such instructions can be divided into the following categories:
   A. Bit operations with an I/O port specified as one of the operands.

B. Data transfer instructions with an I/O port specified as an operand.

C. Test bit and branch on condition with an I/O port or memory location specified as an operand.

D. Single byte subroutine calls via instructions simulating an interrupt.

E. Instructions with implied operands. This includes internal stack instructions and instructions using the A and B registers.

The Solution to the Problem

Before a way for C to give the above advantages can be found, the hard restrictions the language imposes must be stated:

1. Static and Extern class C variables must be stored in contiguous memory.

   A. There is no way to specify the address of a variable from C. Due to the lack of information, compilers allocate variables sequentially in the order of their appearance.

   B. The mode of access for all variables in this class must be the same. The mode of

access for the limited internal RAM is different than for external RAM. Unless the program contains very few variables this generally excludes the internal CPU RAM from being used.

2. Automatic class variables must be implemented on an external stack.

   A. The internal stack overlaps the CPU registers in the internal RAM. The limited space available in this RAM must be reserved for subroutine return addresses stored there by the TMS7000 call instruction.

   B. Frame pointer indexing is required to access automatic variables. It is impossible to do indexed addressing on the internal stack.

3. Register variables can't be associated with any one register. Specifying a variable as belonging to the register class merely suggests that the compiler keep the variable in a register as long as possible. The rules also allow as many register variables as the programmer desires to be specified. No consideration as to the number of physical

registers available is required by the programmer. The only rule governing the compiler's usage of register variables is that ones not assigned to registers must be assigned to automatic storage.

4. All arithmetic in C is done with a minimum precision of 16 bits. Conversion of 8 bit operands to 16 bits is mandatory.

These C language restrictions conflict with the goal of making the code generated by the TMS7000 C compiler both capable of replacing some assembly code section and interfacing with existing assembly code sections. The following design decisions were reached:

1. The rules governing the precision used in arithmetic evaluations would have to be revised to allow more efficient calculation of 8 bit quantities. 8 bit precision will be the most often used precision on the TMS7000.
2. A method must be found to allow registers and ports to be specified in C in the same way as memory locations.
3. The code generator must implement a prioritized variable caching scheme. All variables would be held in the available registers as long as possible. Those variables with the

lowest priority will be the first to be dumped from the registers. Register class variables will have the highest priority.

4. For configurations having no external memory, the programmer will not be allowed to declare any variables. He will be restricted to registers and ports and must use the coding method stated in item 2.

5. Interrupts are a lost cause. C routines used in interrupt routines will require assembly language interfaces.

The next two sections detail the coding method for accessing registers and ports and explain the required changes to the C conversion rules.

Register Access Method

The TMS7000 registers are accessible as memory locations in the 0-FF hex address range. The CPU ports are also accessible as memory locations and their address range is 100-1FF hex. By loading a C pointer variable with the above addresses and applying a unary star operator, access via normal memory addressing instructions can be achieved. Access via special register or port instructions is not possible since the value contained in the pointer variable can't be checked at

compile time. Only pointer constants are known at compile time. Therefore, the only chance the compiler has to recognize a port or register address is if it is specified as a constant using the coding technique demonstrated in Figure 11.

```
*((char *) 0x9f) = value ; /*store to register*/
value = *((char *) 0x9f) ; /*reference register*/
```

Figure 11: TMS7000 Register Access Method

The coding register and port accesses can be simplified using a preprocessor defined macro to name the port or register being accessed. Figure 12 contains the same example as Figure 11 except that a preprocessor macro has simplified coding.

The above method works within the established framework of C and requires no modification of the language. The next section discusses the required modifications to the C conversion rules for 8 bit arithmetic.

Arithmetic Conversion Rules Changes

The smallest arithmetic precision available in the standard C language is 16 bits. All byte (character)

variables are converted to 16 bits when they are loaded into CPU registers. Since all of the TMS7000 instructions work only with 8 bit operands, it would be grossly inefficient to convert an 8 bit operand to 16 bits to perform arithmetic calculations whose results can adequately be contained in 8 bits. The following rules remove this inefficiency while maintaining maximum compatibility with previously written C programs:

1. Unless explicitly casted, constants are assigned the smallest precision their value can be accurately represented in.

```
#define R1    *((char *) 0x9f)

R1 = value ;    /*store to register*/
value = R1 ;    /*reference a register*/
```

Figure 12: Register Access Simplified

2. For binary operators other than multiply, if both operands are byte type then 8 bit arithmetic will be performed and the result will have an 8 bit precision. For those results that possibly may not fit in an 8 bit

precision, the programmer can override this rule by casting either operand to a 16 bit data type.

3. The multiply operator will always generate a result with a minimum precision of 16 bits. This rule occurs because the TMS7000 multiply instruction produces a 16 bit result.

4. For any situations not covered in the above rules, the standard C conversion rules prevail.

This completes the preimplementation design. The next section begins the discussion of the implementation and its ongoing design.

Implementation of the Symbol Table

The symbol table is implemented as an independent modular unit. The issues of what types of symbols were to be represented and the possible links required for the representation of aggregate types and their members are ignored by this unit. They are to be resolved by the routines that call the symbol table routines. All symbol table nodes are considered to be the same size and containing fixed information common to all symbol types. It was assumed that methods allowing additional fields to be added to some nodes would be found later.

This gamble payed off when various forms of node expansion were combined with various methods of storage management via experiments on paper, and two well meshed methods were found.

The symbol table is implemented using the hashed bucket chain shown in Figure 13. The bucket chain allows the symbol table to expand to the limits of available storage using simple coding techniques that ensure a modest retrieval time.

A symbol table as displayed in Figure 13 is created for each scope in existence for the code being read. The scopes are as follows:

1. Global Scope: This scope contains all symbols declared outside any C function or main program.
2. Function Scope: This scope contains all variables which are declared at the beginning of a function.
3. Local Scope: This scope contains all variables which are declared inside a function but not at its beginning. The only such legal declarations are those occurring at the beginning of a curly brace enclosed code block. Multiple local scopes can be created by nesting code blocks within code blocks. The block

having the deepest nest has the most local scope.

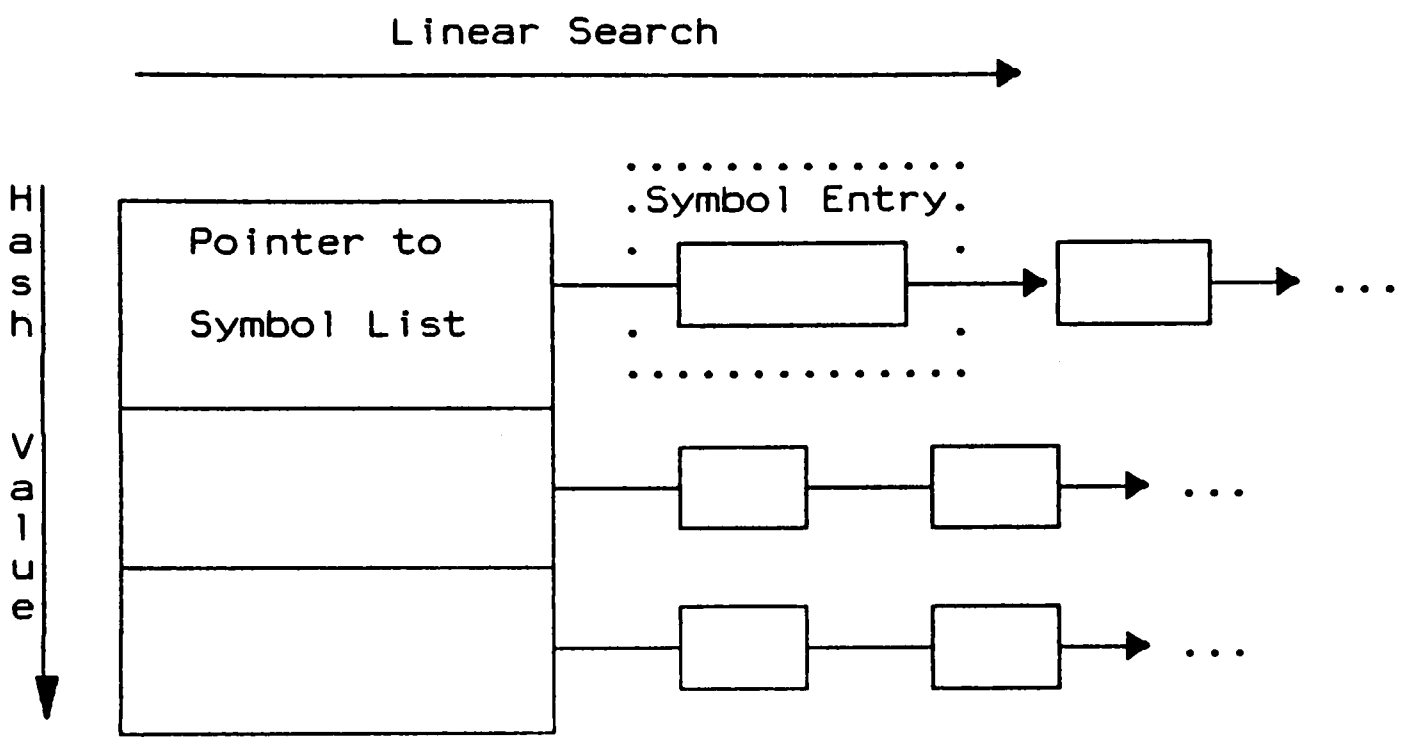


Figure 13: Symbol Table Organization

The symbol tables are linked into a link list as shown in Figure 14. As a scope is entered, a symbol table is created and inserted at the head of the list. As a scope is exited, its symbol table is removed from the head of the list and its storage is freed.

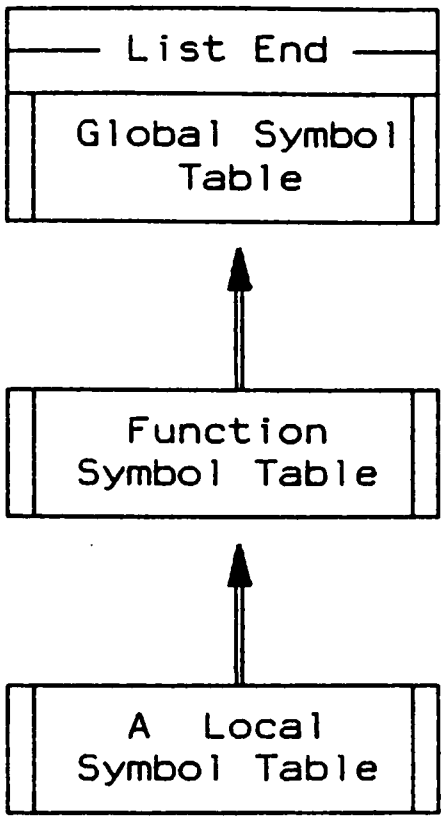


Figure 14: Symbol Table Scoping List

The ordering of the list is significant in that it allows automatic scoping to be performed. To find a symbol, the symbol table routines scan each symbol table in the order of the linked list. The scan stops when the symbol is found or when the search fails in the table at the end of the list.

If the symbol is never found then an undeclared variable error message is printed. To prevent further cascading of error messages, the compiler automatically performs a declaration for the variable. The undeclared variable is given an integer type since a variable of this type is least likely to get into any more trouble.

The definition of a variable symbol only uses the symbol table at the head of the list. That symbol table is checked for the symbol and if the symbol is found, it is doubly declared. Otherwise, the symbol is placed in the symbol table.

Labels are defined and referenced at the function scope level regardless of the scope level it is referenced or declared at. The procedure is different from the one used for a variable due to the fact that unlike variables, forward referencing is allowed for labels. Besides a different procedure, labels also require that their symbol table nodes be marked with an indicator showing their two definition statuses: "referenced"

and "defined." The two statuses are required so that a label which appears in a goto statement but is never defined (never labels any statement) can be detected. Undefined labels are detected when a function level scope is exited. The procedure for referencing a label is as follows:

1. Lookup the label in the function symbol table.
2. If the label is found in the symbol table then goto step 4.
3. Create a symbol table node for the label and mark it referenced.
4. End.

The procedure for defining a label is as follows:

1. Lookup the label in the function symbol table.
2. If the label is not found goto step 6.
3. If the label found is marked referenced, goto step 5.
4. The label is doubly defined. Print an error message and goto step 7.
5. Change the status on the located label from referenced to defined. Goto step 7.

6. Create a symbol table node for the label, mark it defined, and enter it into the symbol table.
7. End.

The function declaration and definition algorithms use a symbol marking procedure similar to that used for labels. This marking has two values: declared and defined. These markings are necessary because any number of declarations may occur but only one definition. Redundant declarations which are consistent have no effect.

A function declaration occurs when a function name occurs in a declaration statement but has no defining block of code following it. A function declaration may occur in any scope. However, the function is declared in the global symbol table. The following list of points outline the declaration algorithm:

1. Lookup the function name in the global symbol table.
2. If the symbol is found, goto step 4.
3. Create a symbol table node for the function symbol, mark the node with a "declared" status, give it the type specified, and enter it into the symbol table. Goto step 5.

4. Compare the type stored in the symbol's node with that of the declaration. If they disagree, print an error message.

5. End.

A function definition occurs when a function declaration in the global scope precedes a code block. A function can only be defined in the global scope since nested function definitions are not allowed in C. The definition may only occur once. The following list of points outline the definition algorithm:

1. Perform the declaration algorithm.

2. If there was an error goto step 5.

3. Check the status of the symbol table node. If its status is "defined," print a duplicate definition error message and goto step 5.

4. Change the symbol's node status from "declared" to "defined."

5. End.

A function may be referenced without ever being declared or defined. When a function reference is detected by the compiler for a function which has not been declared or defined, the compiler does an "automatic declaration." It declares the function to be one returning a simple integer type. If a later declaration

or definition specifies the function to be of a different type, an error message results.

Figure 15 shows an example of compatible definition, declaration, and reference. Figure 16 shows an example of an incompatible definition, declaration, and reference.

So that consistency may be checked, all function symbols are kept in the global symbol table. The following list outlines the reference algorithm:

1. Lookup the function name in the global symbol table.
2. If the function is found, obtain its type information and goto step 4.
3. Create a symbol table node for the function symbol, mark the node with a "declared" status, give it a simple integer type, and enter it into the symbol table.
4. End.

Access to the global scope, function scope, and the most local scope is required at all times. However, not all of these scopes always exist. It was found that by keeping a set of three currency pointers, this complexity could be avoided. These pointers are called the "global currency," "functional currency," and "local currency" pointers. The algorithms above use these

pointers to access the three scopes and are oblivious to the existence or nonexistence of any scope. The algorithm used to maintain the currency pointers is outlined below:

```
char zap() {                    /*definition*/
     return(5) ;
     }

zoz() {
     char ziz() ;               /*declaration*/
     ziz() ;                    /*reference*/
     zap() ;                    /*reference, consistent
                                  because global
                                  definition occurs
                                  before use.*/
     }

char ziz() {                    /*definition*/
     printf("hello world\n") ;
     }
```

Figure 15: Consistent Function Declaration, Definition, and Reference

1. Upon entry into the compiler program, the symbol table for the global scope is created

and all three currency pointers are set so that they point to it. The global scope symbol table is never deleted and the global currency pointer is never changed.

2. Regardless of the previous history of the compiler, if the global scope symbol table is the only one in existence, all three pointers will point to it.

```
zoz() {
     ziz() ;    /*reference, auto declared as
                  an integer function.*/
     }
char ziz() {    /*defined as a character function*/
     printf("hello world\n") ;
     }
```

Figure 16: Inconsistent Function Definition and Reference

3. If the function and global scope are in existence, the function and local currency pointers will point to the function symbol table.

4. Otherwise, the function currency pointer points to the function scope symbol table and the local currency pointer points to the symbol table of the most local scope in existence.

The above discussion is only an outline of how the symbol table is managed in the TMS7000 C compiler. Other aspects of its operation are interwoven with different parts of the of the compiler, and more information about its operation can be found in the following sections.

Implementation of Storage Management

Good storage management is crucial for the compiler to achieve a good execution speed. It should be apparent from the previous section that entering and exiting a scope in C causes a great deal of activity in the storage management routines. What is not apparent is the actual frequency with which C forces this occurrence to take place. In C, every code block that is entered causes a new scope to be created. This means that a new symbol table is created every time a loop, an if-then-else, or a switch containing more than one statement is entered. For the most part, these symbol tables will be empty and in fact, many compilers[15]

attempt to take advantage of this fact by requesting less memory for these symbol tables. However, smaller storage requests do not produce any execution speed benefit. The time spent in storage management routines will generally be proportional to the number of requests and independent of requested storage size (large fixed overhead per request).

Further, experiments on the TI 990 computer with a first-fit memory management scheme showed that programs making large numbers of requests to the storage management routines seemed to have a much slower execution speed than similar programs which made a few large requests. Although no hard timing data was available, coding variations tried in the storage management routines seemed to indicate that searching the memory block list was the principal bottleneck. Because of the large number of requests the C compiler was likely to make, it seemed wise to avoid storage strategies that involved "packet lists" of any kind. The simplest alternative was stack allocation.

Providing the sequentially ordered storage allocation/release imposed by stack allocation can be met, stack allocation is clearly superior to list managed allocation. The justification is simple. If the assumption is made that all blocks are allocated and released

in addressing order, then the free list in the list managed allocation will be a stack. The "stack pointer" for the list method will be the root pointer of the free list. However, if there are two sources of ordered storage allocation and storage release, then the free list in the list method becomes fragmented. A search of the free list is required to obtain each new storage allocation. An insertion sort is required upon each storage release to keep the free list ordered and to coalesce adjacent free areas. In addition to the overhead due to the generality of the list method, the freeing of an entire symbol table or tree becomes more difficult. A symbol table, for example, must be freed node by node since the nodes are not guaranteed to be part of the same contiguous memory block.

In the stack allocation case, two stacks can be created to handle the two different sources. Two stacks can be efficiently implemented using the double ended stack approach[17] shown in Figure 17.

The two sources for storage allocation are expression trees and symbol table nodes. Because tree management and symbol table management algorithms function independently, their storage allocations can not be cooperative.

Stack allocation's requirement of sequential allocation/release means that the following restrictions must be imposed:

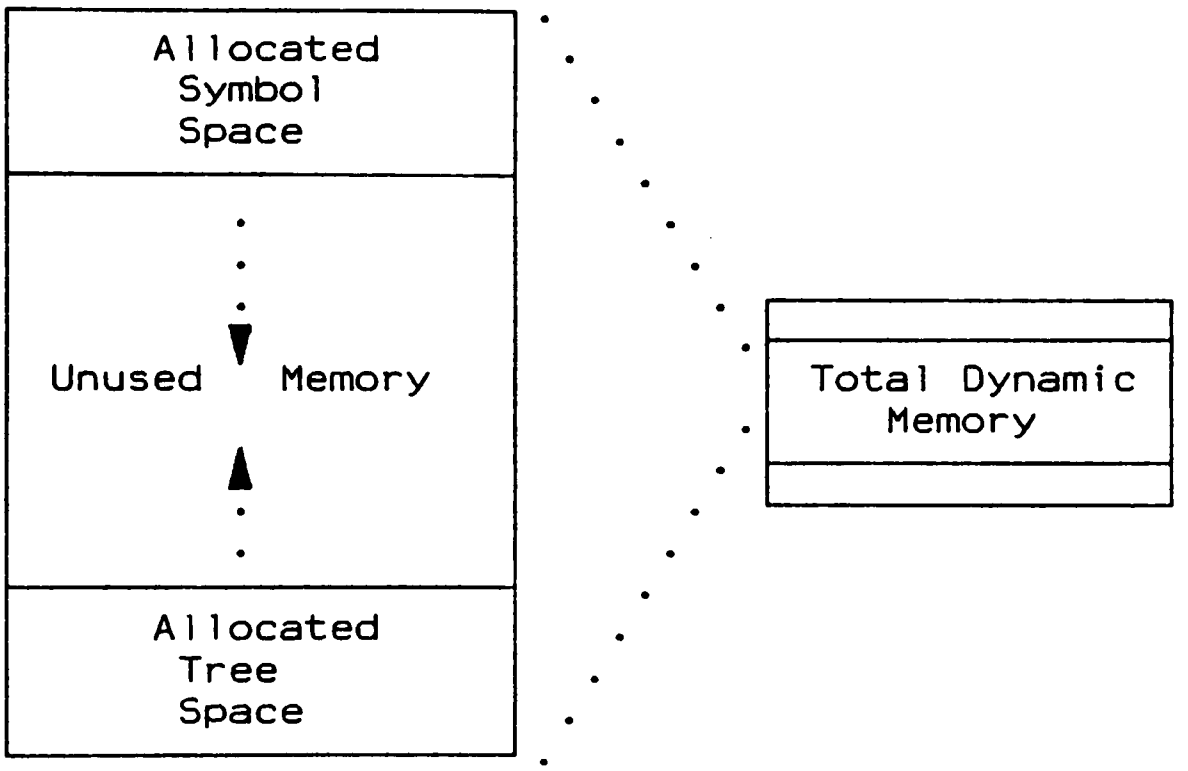


Figure 17: Storage Management Scheme

1. No variable must have a lifetime beyond the life of its scope. This is to allow the variables of an entire scope to be freed via resetting the allocation stack pointer.

2. No variables of a scope higher or lower than the current most local scope could be allocated.
3. The life of a tree node or subexpression tree could not exceed the life of the expression to which they belong.
4. The need for incidental dynamic storage by other parts of the compiler is considered too small in comparison to symbol table and tree needs. It must be handled by a small separate pool of memory or statically allocated via declaration of the data structure. This restriction allows access to the stack storage management to be strictly controlled and the address sequential nature of memory management requests to be preserved.

Once the stack method of storage management was adopted, several opportunistic optimizations could be applied to increase the compiler's efficiency. They principally centered on coupling some aspects of storage management and symbol table management, and can be stated as follows:

1. The use of stack allocation allows symbol table nodes to be expanded by simple physical

extension. Symbol nodes can be extended by a second call to the allocation routine.

2. The scope creation and termination routines can automatically manage storage reclamation and initialization. Symbol tables for each scope are represented by a data structure containing the hash pointer array and a pointer variable which points to the symbol table of the next higher scope. By including the stack pointers for symbol and tree stacks in this data structure, automatic storage management can be achieved. The allocation routines use the stack pointers in the most local symbol table. The most local symbol table can quickly be reached via the "local scope currency" pointer. When a scope is created, its stack pointers are initialized to the value of the previous scope's stack pointers. When a scope is exited, the data structure for its symbol table is simply unlinked from the scoping list and the local scope currency pointer is updated. This effectively frees all storage in the exited scope. Thus, storage management is performed automatically.

One particularly troublesome aspect of the stack method of storage management is that if it were implemented exactly as stated, allocation of symbol nodes for symbols in a scope more global than the current one would be impossible. This is an important point because label and function declarations must be entered in a specific scope's symbol table regardless of what the current scope is. The first prototype's solution was to violate the original C specifications and force all declarations to be entered into the current local scope's symbol table.

This seemed to be advantageous in the case of statement labels. In the original implementation of C, a branch via a goto to a label in a more local code block was possible. This direct branch into the more local block caused the block's local variable activation code to be skipped. The rules address this problem by allowing this kind of goto to be legal but specify that any initializations specified in the variables' declarations are not required to be performed. This kind of operation did not seem very desirable. By enforcing the scoping of labels the compiler (and possibly the programmer) would operate more efficiently.

Although the first prototype compiler initially used the same approach for function declarations as for

labels (i.e. declaring them in the current scope), undesirable aspects surfacing later forced a change. Initially, if no global declaration existed for a function about to be referenced or declared, the function was declared in the current most local scope. This prevented later definitions from being verified for consistency in type against previous references of the function. The worst error that could occur would be truncation of the return value. Later, when the rules were changed to allow a smaller than default precision to be placed on the stack, a type mismatch could result in garbage values being returned.

At first, the removal of this error checking feature seemed to be a minor point. This type consistency check only works if the subroutine definition/declaration and reference appear in the same file. Functions defined in other files and never declared in the current file could not be checked in any case.

After consulting with engineers involved with the TMS7000, it was discovered that many TMS7000 assembly language programmers place an entire program's source code in a single file. For these reasons, the problem had to be solved. To solve this problem, functions must always be declared in the global scope symbol table as

specified by the Kernighan and Ritchie rules so that their consistency may be checked.

The problem with implementing this rule is that it can cause a conflict with stack allocation requirements. One such requirement is that all global symbol nodes must be part of the same contiguous segment of memory on the allocation stack. This would normally translate into the requirement that all global symbols must be allocated from declarations within the global scope. For example, if a function scope with local variable declarations is entered and then a function reference is encountered, the resulting function name symbol will be separated from its fellow global symbols by the more local symbol nodes allocated before it. The longer lived function name symbol will prevent those local variables from being freed upon exit from their scope. Freeing them (resetting the stack pointer) would also free the function name symbol node. This causes a "dead zone" of permanently useless space on the stack in the area where the local variables once resided. The dead zone is illustrated in Figure 18.

Since global symbols are never freed, this stack zone remains useless unless some form of stack compaction, such as the In Situ technique outlined in Standish[17], is implemented. Stack compaction requires

recopying some of the global symbol nodes to new positions. This method would have to be applied upon completion of a function's translation but before the function's symbol table was released.

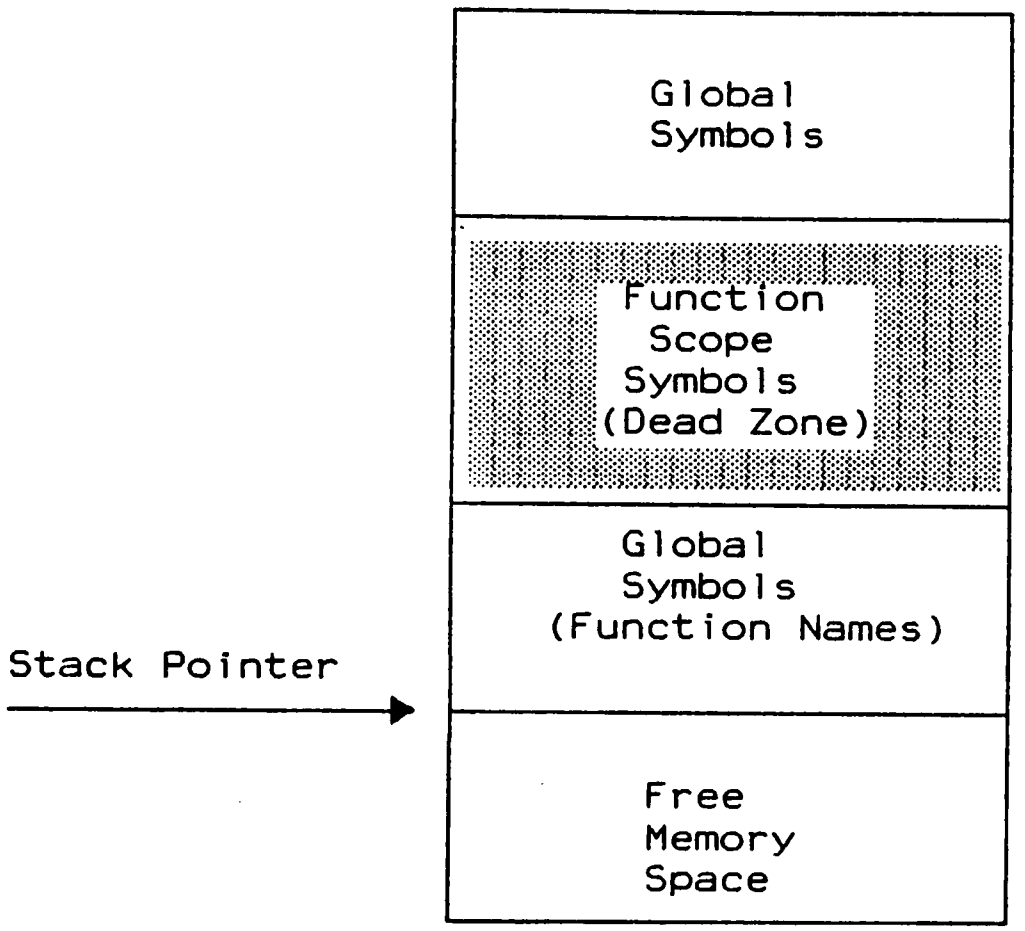


Figure 18: Illustration of Dead Stack Zone

The net result is that 8 times more operations are done than maintaining the free list in the list managed

implementation. One solution to this problem which avoids the problems of compaction is to allocate a "stack margin" prior to entering the function scope as shown in Figure 19.

This stack margin consists of free space reserved for the global symbol table while symbol tables more local in scope are active. When an undefined and undeclared function is referenced within a function, or when a function is declared within a function, space for the symbol table node containing this global function name is allocated from the stack margin. Thus the stack margin size must be large enough to house the maximum number of global symbols likely to be declared from within a more local scope. The actual size of the stack margin is by nature a guessed value based on an assumption of the programmer's coding style. The effect of the actual size used is minimized by the allocation of a full sized stack margin upon every entry to a function scope. No space is permanently unused since any free space remaining in the margin is totally reclaimed upon exit from a function's scope. Thus, this method produces no permanent "dead zones."

The stack margin is a simulation of a third stack. The optimal way to implement storage management would be to have a third stack for function and local symbol

table symbols. This would remove the arbitrary limits imposed by the stack margin method and greatly simplify storage management. A typical implementation of a third stack, such as Garwick's Technique outlined in Standish[17], would require a separate memory segment for the stack. A separate segment is required because it is the only way to keep the addresses stored on the stack relocatable.

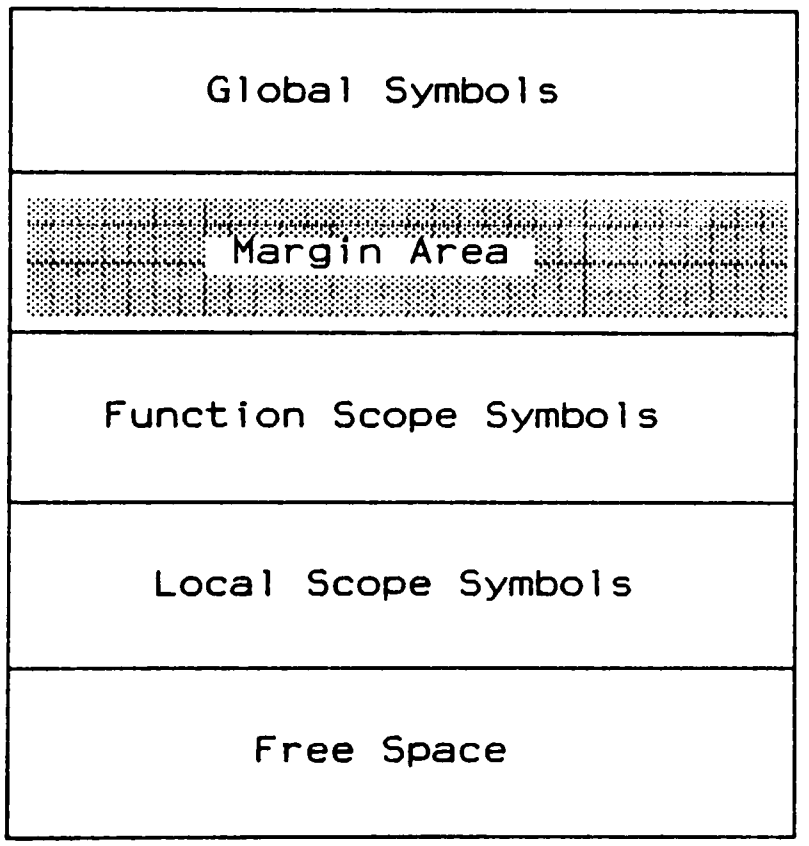


Figure 19: Illustration of Stack Margin Area

Third stack techniques require a stack copy upon stack grow/compress. The C compiler for 8086 microprocessor upon which the cross compiler is implemented will not allow a second data segment area. This is due to C's lack of address space control.

Design of the C Lexical Analyzer

The TMS7000 C Compiler's lexical analyzer was the first section of the compiler to be designed, implemented, and tested. The lexical analyzer, as shown in Figure 20, serves as the major connecting link between the various major sections of the first phase. Its design was critical in that it determined the shape and form of the statement parser, symbol table manager, and expression analyzer. Although tests of coded sections of the lexical analyzer removed most design flaws, the true evaluation of the design could only come after the statement parser, symbol table manager, and expression analyzer were designed. A good lexical analyzer design would minimize the complexity of the design for the other three sections.

The first difficulty to be overcome in the design involves deciding what to do with multicharacter operator symbols.

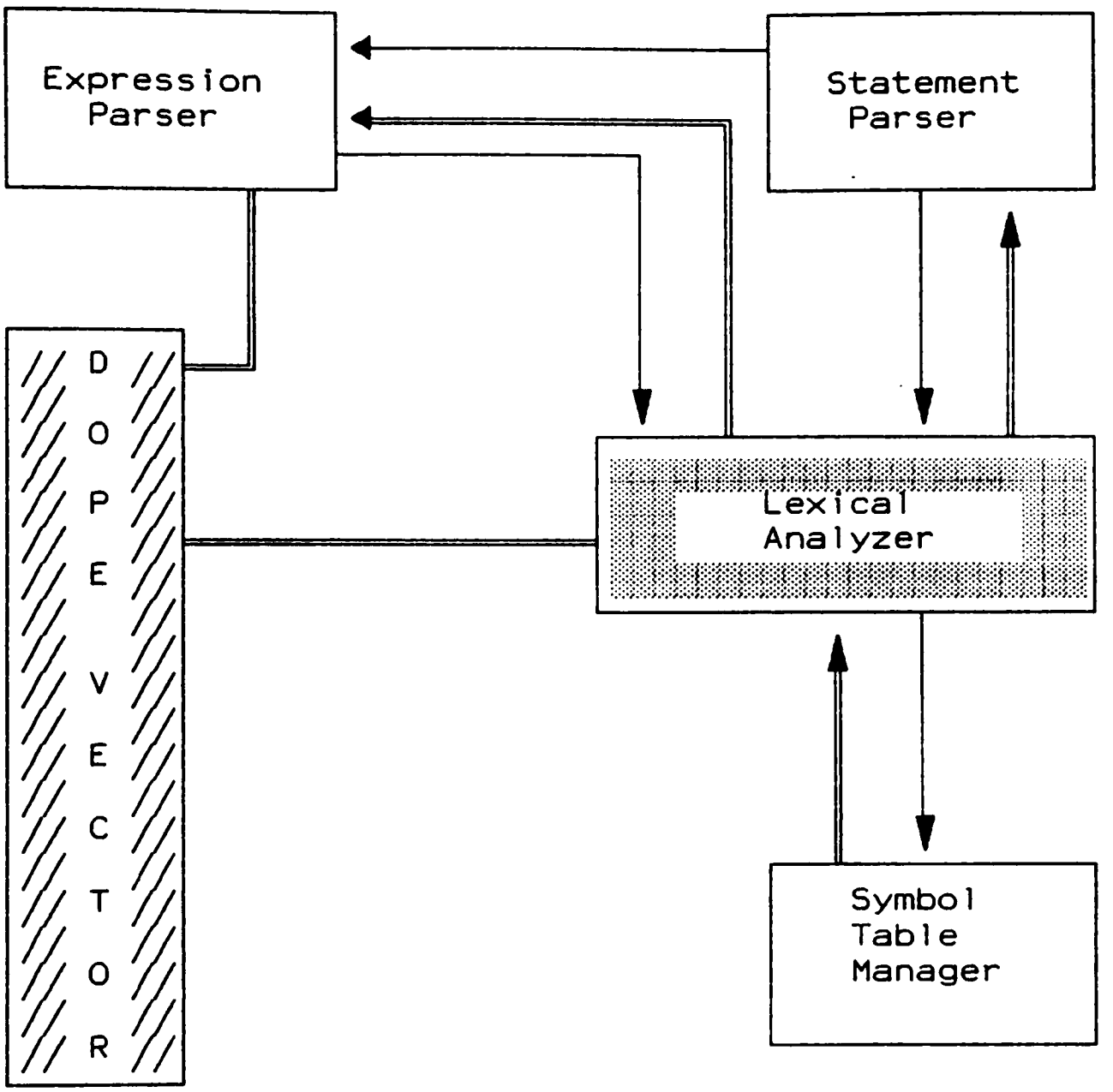


Figure 20: Organization of the Compiler's First Half

In many languages such as Pascal, this level is also trivial because multicharacter operators are recognizable as soon as enough characters are collected.

However, a collection of adjacent characters in C can be interpreted as several different operators according to the C grammatical specification. The lexical analyzer must deal with this ambiguity by a having a set of special case rules outside the stated BNF specification. Figure 21 shows a comparison of Pascal with C.

Pascal expressions can be unambiguously interpreted directly from the BNF rules. This is due to the fact that every arithmetic operator must be separated by a value, variable, or parenthetically enclosed expression[19].

C has prefix and postfix unary operators. Not all operators are separated by values. In particular, unary and binary operators may be adjacent and multiple unary operators may be applied to a value. Some C unary operators are composed of binary operators placed adjacent to each other. There are two choices for resolving this grammatical ambiguity:

1. Require explicit delimiting of operators via white space or values.
2. Identify an operator by reading the longest string of characters that can make a legal C operator. This is the rule chosen by the standard.

Language Example	Explanation
Pascal	
A = B * -5	This is illegal. Two arithmetic operators are adjacent.
A = B * (-5)	This is legal.
A = -B * (-5)	This is also legal. Assignment operators, and relational operators can be adjacent to arithmetic operators.
C Language	
c= a+++b ;	Could be c = a++ + b or c = a + ++b. C interprets this as c = a++ + b.
c = a+--b ;	This is c = a + --b. C interprets it correctly.
c-->>= b ;	An example of a 3 character operator.

Figure 21: Comparison of Unambiguous and Ambiguous Syntax

This rule resolves the ambiguity in the first C statement of Figure 21. The rule also allows recognition of statement 2 correctly. Statement 3 shows how the lexical analyzer may have to inspect up to 3 operator symbols to know what operator token has been recognized.

The next and most difficult step in the design was to decide how much operator overloading should be removed in the lexical analyzer via removal of context dependencies. The two extremes in design philosophy are as follows:

1. Context independent tokens. The name of the token depends only on the sequence of characters recognized. This produces a very simple lexical analyzer but places a heavier burden on the parser and complicates the specification of the grammar to the parser.
2. Context dependent tokens. The name depends on the sequence of characters and the context the sequence occurred in. This produces a complex lexical analyzer but results in a context free language being passed to the parser. The grammar specification becomes much simpler.

Method 2 was finally chosen based on the following considerations:

1. The structure of the code of a top-down parser exactly mimics the BNF grammar specification. A simple grammar means a simple code organization.
2. Redundant BNF clauses are required to impose context dependent restrictions. For example, to forbid the use of the comma operator in an expression for a single context means that both contexts must have a BNF specification for expressions. In both specifications, the clauses would be identical except that the comma clause would be missing for the restricted context's specification.
3. Actual specification of the C grammar to the parser requires some degree of experimentation. This was deduced from the test history of the TI 990 C Compiler. In bottom up compilers such as that one, the grammar change could be made by changing the grammar table fed to the parser generator program. In a top-down compiler, a grammar change involves discarding sections of compiler code and writing new replacement sections. Context

independent BNF requires fewer changes because it can more exactly specify the language.

Context independent BNF does not eliminate all context dependencies in the BNF specification but just eliminates context induced redundant clauses. Removing redundant clauses makes changes easier for the human composer to follow. The language becomes as easy to specify as one without overloaded operators.

Implementing a lexical analyzer using method 2 requires communication between the lexical analyzer, parser, expression parser, and symbol table manager which is intimate yet maintains the modularity of the individual sections. The interface specified was a 32 bit flag word, Γ, to contain context information, a dual purpose 16 bit token id value/token node pointer, and a shared token database (dope vector), Φ. Using Γ and $\Phi(\Gamma,\Upsilon)$, where Υ is the current token, an extension of the Bounded Context approach[19] was used for lexical analysis. The Bounded Context approach, used mainly for expression parsing, used a partitioned table of productions with each set of productions being grouped according to context. The proper set was chosen by choosing the context currently active in an expression. Because the application here is lexical analysis, not

parsing, productions were not used. Instead, Γ and each of its bits containing context information, imposed a partitioning upon the token recognition code itself. Γ extends the Bounded Context approach because instead of a single context being active at a time, a set of overlapping contexts with differing bounds may be simultaneously active. The role of ∉(Γ,τ) is to provide a legality check of τ for the current context and to perform a next state calculation for Γ.

Γ also plays a primary role in recognition of context dependent overloaded operators (τ(Γ,θ), θ = sequence of input characters). Such operators include prefix/postfix context dependent operators and operators with expression/nonexpression context dependencies. An example of a expression/nonexpression context dependent operator, the comma operator, is shown in Figure 22. In an expression, a comma is a binary value returning operator. Outside of an expression, it is a separator. In the function call, unless enclosed within a parameter expressions parenthesis, the comma is taken to be in a nonexpression context.

Another role Γ plays is operator partitioning. Constant expressions only allow a subset of expression operators and value tokens to be used. By using Γ

partitioning, a set of duplicated productions for expressions and constant expressions can be eliminated. The production for constant expression is reduced to:

CONSTANT_EXPRESSION ⇒ EXPRESSION

with an associated action to set Γ(Constant) context.

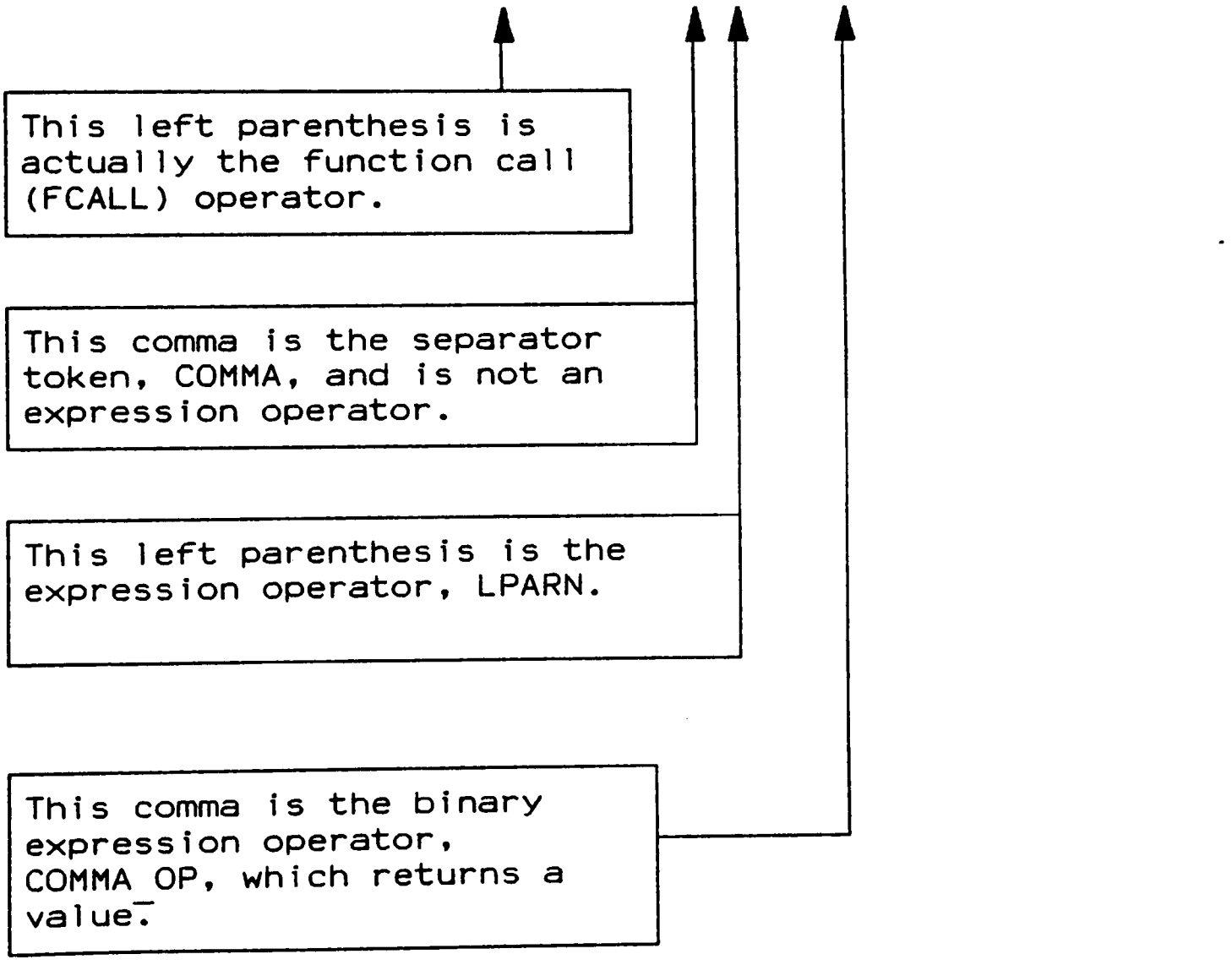


Figure 22: Context Induced Difficulties

The constant expression operator subset is then imposed by the lexical analyzer.

The last service provided by the lexical analyzer is the introduction of "synthetic tokens" ($T(\theta,T) : \theta \in \emptyset$). These tokens do not correspond to any characters in the input stream but their injection into the parser input stream in certain context situations greatly simplifies statement and expression parser coding. An example of this is the PARM token. This token is a do nothing token used to bind parameter expression trees as subtrees to the main expression's tree. A unified tree allows more streamlined code generation via tree walks. This completes the description of the lexical analyzer.

The Implementation of Structures and Unions

How a structure is implemented in a compiler depends on the implementer's view about how a structure should be used. In current C implementations, these views fall into two philosophical categories. They are as follows:

1. Members of a structure are similar to members of a set. Under this philosophy, every structure has its own name space, and identically named members may be declared as long as they

are not contained in the same structure. Members must be qualified by a structure variable so that their owning hierarchy can be determined. The owning hierarchy must be known so that its position in the hierarchy can be determined. Lattice C and several other compilers adopted this philosophy because it is more advantageous to application programmers.

2. Members of a structure are symbolic ways of representing offsets from a base address pointer. Under this philosophy, the members of the structure may be applied to any storage area, whether it be a structure variable, an integer variable, or a set of memory mapped registers. Thus, all structure members belong to the same name space. This philosophy imposes a mandatory rule that if any identically named members are declared, the members must be declared in such a way that they all have the same address offset. The C language specification in the book by Kernighan and Ritchie[8] and the early Unix C compilers adopted this philosophy.

Since microprocessor applications demand low level access, the second philosophy was chosen. This method of implementation allows a clean representation of assembly code's indexed addressing. Although method one has stronger typing and is advantageous to application programmers, control programmers would often be forced to override the typing due to their requirement of flexible hardware access.

To demonstrate how structures were implemented, it is necessary to define some terminology. The sample structure declarations in Figure 23 will be used in the explanation:

```
struct tag_name_1  {
      int member_1 ;
      char member_2 ;
      struct tag_name_2 member_3 ;
      long member_4 ;
      }  var_name_1 ;


struct tag_name_1 var_name_2 ;
```

Figure 23: Definition of Structure Terminology

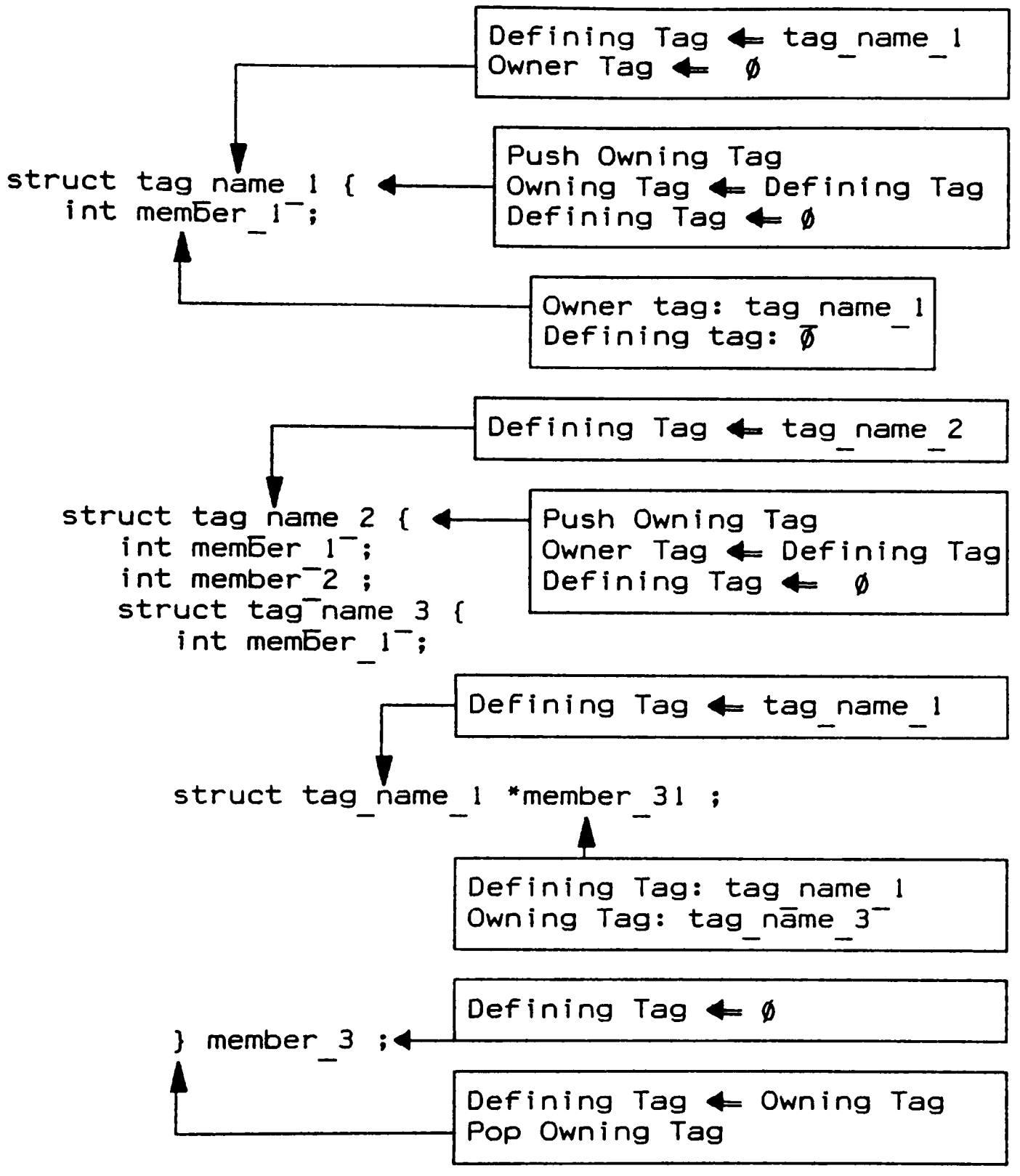
In Figure 23, tag_name_1 is the name which uniquely identifies the collection or structure type being

defined. Tag_name_1 is the type name or "tag" of the structure. For both var_name_1 and var_name_2, tag_name_1 is the "defining tag." For each of the members of the structure, tag_name_1 is the "owning tag" or the tag identifying the collection to which the members belong. Member_3 is a particularly interesting case since it has both a defining tag, tag_name_2, and an owning tag, tag_name_1.

Since member_3 is an aggregate type (structure variable), its symbol table entry must contain a pointer to the type that defines it so that its size may be determined. As a result, all aggregates have a defining tag. Since only one definition of the aggregate is allowed, every aggregate has only one defining tag.

Figure 24 shows a typical recursive structure declaration. The following points may be observed from the figure:

1. Any nonfunction declaration and function pointer declaration used outside a structure may also be used inside a structure. Thus, the same routine may be used to parse both.
2. Because of the recursive nature of the declarations, a set of currency pointers to the current owning and defining tags' symbol table node must be kept.




```
struct tag_name_1 {
   int member_1;
struct tag_name_2 {
   int member_1;
   int member_2 ;
   struct tag_name_3 {
      int member_1;
struct tag_name_1 *member_31 ;
} member_3 ;
   } member_12 ;
int member_T3 ;
struct tag_name_3 member_14 ;
struct tag_name_3 *member_15 ;
} var_name_1 ;
```

Figure 24: Nested Structure Declarations

These pointers must be stacked as the recursion descends and restored from the stack as the recursion unwinds.

By simply copying the value of the currency pointers into the corresponding symbol table node fields of the variable being defined, the first prototype was able to store the complete type information about the aggregate member or variable being defined. As development proceeded beyond the first prototype, it became apparent that the storage of complete typing information alone was not sufficient to allow errors deep within C's recursive declarations to be detected. Recursion induced difficulties in checking and analyzing declarations is a problem somewhat unique to the C language. Recursive declarations are required in C because C does not allow forward referencing of structure types.

The problems can be divided into two categories: detection of endlessly recursive structure definitions and problems with correct member offset calculations in highly recursive declarations.

Member offset calculations can become extremely complex even with a small declaration. Figure 25 shows one of these difficult declarations. This declaration was particularly devastating to the first prototype's

algorithm. The first prototype calculated a member's offset at the time the member was encountered in the source code. The offset was the sum of sizes of each previously seen member. However, the declaration in Figure 25 has members whose size can not be determined at the time they are seen. Their size is unknown because the definition of the member's type is incomplete.

```
struct tag_1 {
        int a;
        struct tag_2 {
                int b;
                struct tag_1 c ;
                } *d ;
        int e ;
        } f ;


main() {

printf("\nThe size of tag_1 is %d.\n",
                sizeof(struct tag_1)) ;

printf("\nThe size of tag_2 is %d.\n",
                sizeof(struct tag_2)) ;

}
```

Figure 25: Member Offset Calculation Test Program

In order to confirm the legality of the declaration and to try and discover other compilers' algorithms, the program in Figure 25 was compiled and executed by several common and well known C compilers. Provided that no diagnostic error messages are generated, the result of the program should be that tag_2 has a size greater than that of tag_1. This result should occur regardless of the machine, implementation, or type sizes of the implementation.

Lattice C was the first compiler tried. The compilation of the test program produced no diagnostic error or warning messages. In this compiler's view, the declaration was correct. However, the results obtained by executing the program were incorrect. Lattice C had generated incorrect code for the program. The results tended to suggest that the compiler had totally ignored the declaration of the variable c.

Desmet C was tried next. The compilation produced no warning or error messages. The program was then executed. Again, incorrect code was generated. The results were different, though, from those of Lattice C. The results agreed with what would be obtained by the first prototype's algorithm. The results indicated that the size of the tag_1 structure at the time of variable c's

definition was used. This means that variable, c, would not contain space for the members d and e.

The last compiler tried was the C compiler delivered with Berkeley UNIX. The compilation produced an error message saying that the size of tag_1 was unknown.

The common denominator between all of the tests and the first prototype is that they generated the member offsets as the member names were parsed. The net result is that an algorithm needs to be found that will delay presentation of the members to the parser and present members to the parser in the proper order for correct offset calculation. Based on the results of the tests, this hypothetical algorithm is evidentially not well known.

Most languages avoid recursive declarations by allowing forward declarations of pointers to structures (records) and preventing the naming of type definitions occurring inside a structure. Pascal, for example, explicitly prevents record type definitions from being nested[19]. This totally eliminates the possibility of mutually recursive type definitions.

Languages which do not allow recursive definitions possess straight forward and well known structure definition translation algorithms such as one derived

by Knuth[16]. C does not satisfy these restrictions and requires a more complex algorithm. The first prototype used a simple approach. It did not allow a definition tag to match an owner tag for nonpointer variables. This method catches simple mistakes but does not catch the endless recursion shown in declaration 1 of Figure 26.

```
struct tag_1 {                          /*declaration 1*/
     int a;
     struct tag_2 {
          int b;
          struct tag_1 c ;
          } d ;
     int e ;
     } f ;

struct tag_1 {                          /*declaration 2*/
     int a;
     struct tag_2 {
          int b;
          struct tag_1 c ;
          } *d ;
     int e ;
     } f ;
```

Figure 26: Detection of Endlessly Recursive Structure Declarations

One might conclude that the first prototype's rule might be corrected by not allowing any nonpointer variable's definition tag to match any previously seen

owner tag whose definition is incomplete. Declaration 2 proves that this rule would not be adequate. The rule would declare declaration 2 to be endlessly recursive since the definition of tag_1 is incomplete at the point where the variable c is declared. The declaration does possess finite recursion and the size of every tag and member size can be determined. Thus, a method for detailed analysis of the relationship between a type's (tag's) and a variable's definition must be found.

The easiest way to perform the required checks is to construct a graph representing the relationships between the definitions of the tags of the declaration. The algorithm for producing the graph should generate a graph which facilitates the checks that have to be made (i.e., for endlessly recursive definitions). This is in contrast to the first prototype's storage of the relationship information in the symbol table, where the primary consideration was facilitating symbol look up. In a properly constructed graph, an endlessly recursive declaration would produce a loop in the graph. The loop represents a circular type dependency. An exhaustive search of an arbitrary graph for loops would be time consuming. For this reason, a set of heuristically derived construction rules for the graph were found which restricts the graph to a binary tree with bidirectional

links. A circular type dependency translates into a subtree whose root node is identical to one of its leaves. The tree constructed will be referred to as the "Record Analysis Tree" (RAT). Each node of the RAT contains the following fields:

1. Pointer to the parent node. The meaning of this pointer will be explained later.
2. Information Field: Pointer to the symbol table node for the tag this node represents.
3. Pointer to left son.
4. Pointer to right son.

Each of the pointer fields are initially set to null when a node is linked into the tree. The net result is that the forward links always point to a subtree if one exists but a subtree may be isolated from its parent tree via a null parent pointer at the subtree's root node. Thus, circular type dependency is encoded by selectively defining or not defining the parent pointer. By performing a traversal via parent pointers from each leaf to a subtree's root (node with a null parent pointer), circular dependencies can be identified. If a duplicate pair of nodes is contained within any traversal, a circular dependency has been detected.

The following discussion describes the rules themselves. Rather than just listing the rules alone, the list of rules is combined with a list of properties designed to make the rules easier to follow. The first point is as follows:

1. The basic idea of the algorithm is as follows:

    A. Tags cause a node representing the tag to be added to the RAT. The RAT node contains a pointer to the tag's symbol table entry. Each mention of a tag in a declaration produces a corresponding RAT node.

    B. Whether the aggregate variable being defined is a pointer or not determines whether or not its defining tag's RAT node's parent pointer is set. Nonpointers set the RAT node's parent pointer.

Each mention of a tag produces a RAT node containing a pointer to the tag's symbol table node. Only the single definition of a tag produces a symbol table node. Thus the RAT serves as a "many to one" mapping between the parsing algorithms and the symbol table. This means that the owner tag and definition tag

currency pointers which formerly pointed to the tag's symbol table node now point to the corresponding RAT node.

The next rule governs the ordering of the nodes in the tree. The basic idea is that given any node in the tree, a list of tags at the same nesting level in the declaration can be obtained by traversing down using the right forward pointers. Tags attached via the left forward pointers are declared at a higher nesting level, i.e., within the current substructure. Figure 27 shows a skeleton declaration and the RAT it produces. Only the forward links are shown for the sake of clarity. The RAT in the figure shows how the RAT exactly mimics the structure of the declaration.

The basic rule which governs that ordering in the RAT and created the tree in Figure 27 is as follows:

2. When a tag is encountered, its representative RAT node is created and linked to the rightmost position in the current owner tag's (pointed to by the owner tag currency pointer) left subtree. This means the following:
   A. Examine the left pointer of the RAT node pointed to by the owner tag currency pointer. If this pointer is null, link

the new RAT node into the tree via the left pointer.

B. If the left pointer was not null then examine the node it points to (call this node α).

```
struct TAG_A {
     struct TAG_B {

          struct TAG_C {
               .......
               } VARIABLE_C ;

          struct TAG_D {
               .......
               } VARIABLE_D ;

          struct TAG_E {
               .......
               } VARIABLE_E ;

          } VARIABLE_B ;

     struct TAG_F {
          .......
          } VARIABLE_F ;

     struct TAG_G {
          .......
          } VARIABLE_G ;

     } VARIABLE_A ;
```

Figure 27: Structure of a Record Analysis Tree
Part 1

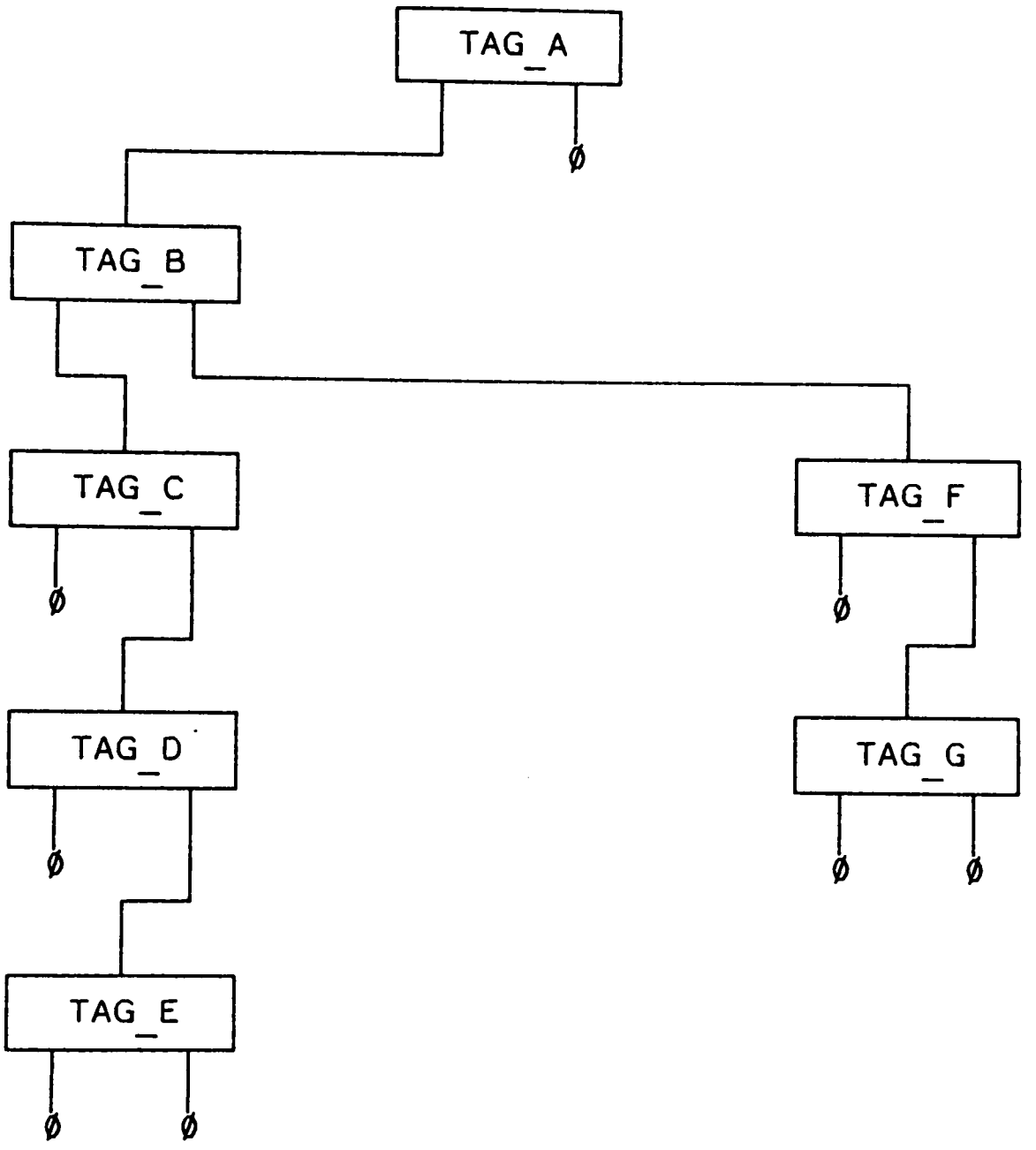


Figure 27: Structure of a Record Analysis Tree
Part 2

a.) If the right pointer of α is null, then link the new node into the tree as the right son of α.

b.) If the right pointer of α is not null, then find the right most node of α's subtree (call this node β). Link the new RAT node as the right son of β.

The ordering of the RAT could be considerably more lenient if it was used only for the endless recursion check. However, the ordering induced by this rule also allows a inorder traversal of the tree to present all of the members to the parser in the proper order for member offset calculation. The next rule is the one essential to the endless recursion check.

3. When a member that is also an aggregate is encountered, the RAT node pointed to by the definition tag currency pointer has its backward pointer set equal to the owner tag currency pointer.

What the above rule effectively does is link a subtree containing a complete type definition to a parent tree via its backward pointer. This parent tree contains a type definition whose size is dependent on the size of the type defined in the subtree.

A tree built according to the rules above allows a structure declaration to be quickly analyzed using a simple analysis algorithm. The algorithm is outlined in the following set of points:

1. The tree is traversed using a inorder traversal.
2. Each node visited whose backward pointer is not null is analyzed by a traversal via the backward pointers to the root node or first node with a null backward pointer (list end). If any node in the list has an information field which matches the value of the information field in the node being analyzed, then the entire declaration is endlessly recursive.
3. When a RAT node is visited, the symbol table node of the tag it represents is checked to see if its size (the size of the type it names) has been defined (size field is nonzero). If the size field is zero then a member offset calculation takes place. Because of the ordering imposed by the traversal, all type sizes required to calculate the size of the type under consideration are known. The members owned by a tag are located by a

"member list." The head pointer for this list is located in the tag's symbol table node. The member offset calculation is performed by scanning the member list and performing the following steps for each member:

A. The current member's offset is contained in the size field of the owner tag's symbol table node. If this is the first declaration of the member (among the entire program's declarations), then the size field is copied into the offset field of the member's symbol table node. Otherwise, the member is multiply owned and the tag size and member offset fields are compared. If they differ, the member occurs at different offsets in two different structure definitions and is thus erroneously declared. An error is reported and the entire RAT algorithm terminates.

B. The member's owner tag's symbol table node's size field is incremented by the size of the member.

In order for the RAT to be used by the member offset calculation algorithm, a "member list" must be

built and attached to each tag's symbol table node. This member list is a list of all members owned by the tag. Because a member can have multiple ownership, i.e., be part of more than one structure, the member list can not contain the member's themselves. Instead, an intermediate member list node is used. The node contains a forward pointer for the list and a pointer to the member's symbol table entry. Figure 28 shows an example of a multiply owned member in a declaration and the associated member list.

The relationships expressed in the RAT only need to be known during the correctness verification of a declaration and the calculation of a member's offset. Once these two steps have been completed, all that is required to be stored in the symbol table is the identifier's name, whether it defines real storage or is a type name, the size of the storage defined, and if it is an aggregate member, its byte offset from the start of the aggregate. This transitory nature of the RAT is exactly analogous to an expression tree. Thus, RAT nodes are allocated from the expression tree memory space. Allocating from the expression tree memory allows a RAT to be freed at the end of each declaration. This allows memory to be conserved.

```
struct TAG_1 {
      int MEMBER_A ;
      char *MEMBER_B ;      /*The member in common.*/
      long MEMBER_C[10] ;
      } ;

struct TAG_2 {
      char MEMBER_D ;
      char MEMBER_E ;
      char *MEMBER_B ;      /*The member in common.*/
      } ;
```

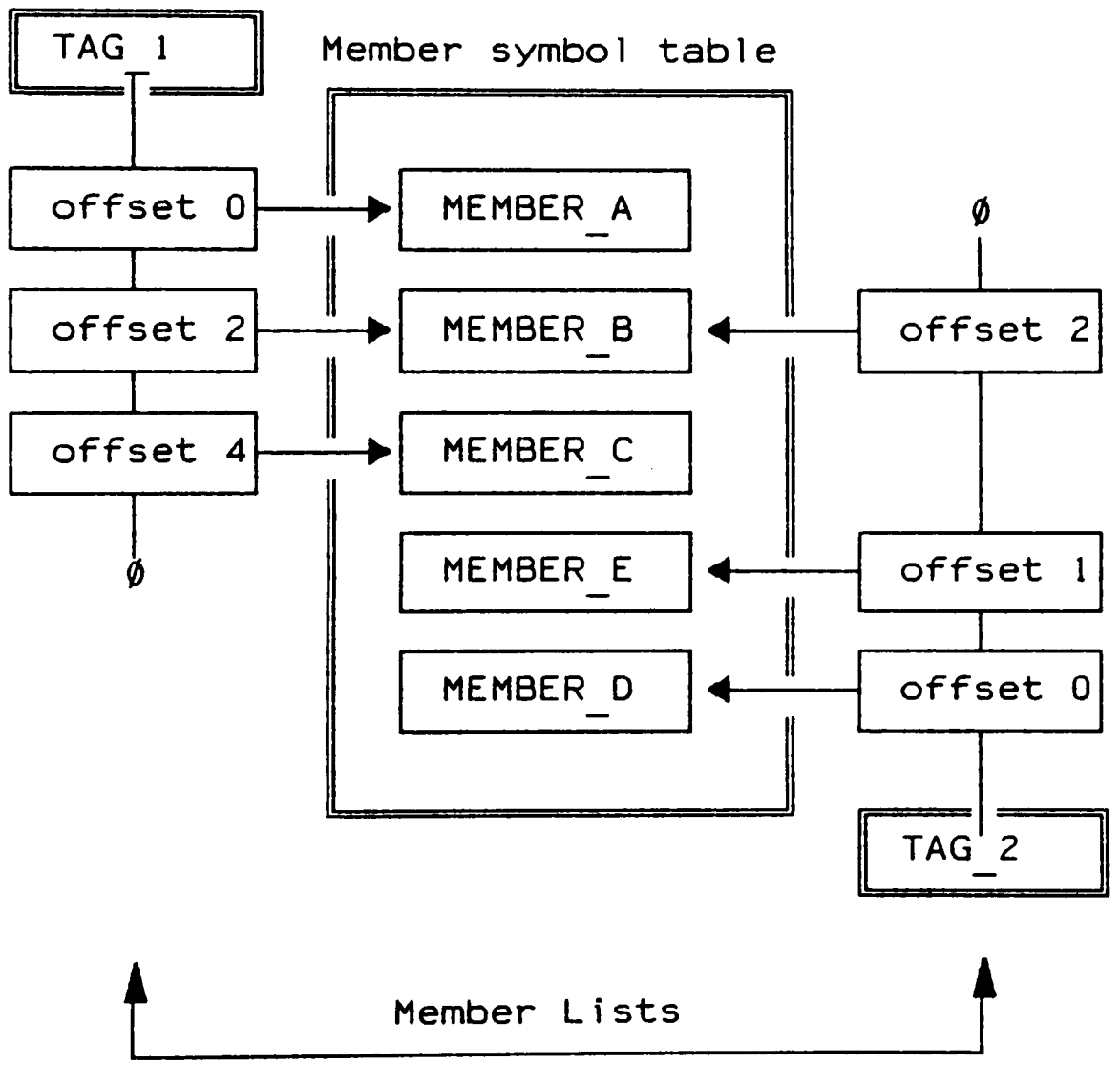


Figure 28: Member List Example

Member list nodes are allocated from the symbol space, however. They are attached to a structure tag's symbol table node prior to the deletion of the RAT. The possibility of a future declaration of an initialized aggregate typed by the currently declared tag forces the member list to be retained. The members for an initialized aggregate must be retained so that the precision required to store each initializer can be determined.

The RAT algorithm was never coded during the compiler project. It was tested on paper for some of the particularly nasty examples shown in this section. This completes the description of the implementation of structure and unions.

Implementation of Symbol Indirection

Indirect referencing from memory and storage to memory (indirection) is one of the main strengths of the C language. Unlike most high level languages, C allows access to the processor's memory addressing modes equal to that of native assembly.

More importantly, C provides an abstraction for this process of indirection. A long sequential chain of assembly indirect reference instructions leading from an initial address to a final value can be represented

by a compact symbolic notation. The same is true for a chain of storage instructions. A self-consistent organization is imposed by this abstraction in that data objects and address objects always remain distinct, separate entities and are never interchangeable. Declarations and abstract data typing operators (casts) permit the allowed uses of an address to be specified throughout the entire indirection chain. The positioning of indirection operators in a declaration or cast and in the actual reference or storage chain of instructions is exactly the same.

C indirection typing is totally distinct from C storage typing. An object in C is specified by 3 orthogonal coordinates:

1. Storage Type - Storage type consists of the following components:
   A. Memory Class - Automatic, static, and extern.
   B. Arithmetic Type - Signed, unsigned.
   C. Storage Precision - Char, short, int, and long.
2. Memory Address.
3. Indirection - Each indirection type consists of a basic indirection mode combined with a storage attribute for the address.

A. Indirection mode:

a.) Pointer to pointer.

b.) Pointer to function.

c.) Pointer to data object.

d.) Data object.

B. Storage attribute:

a.) Constant

b.) Lvalue

Table 3 shows the indirection types resulting from the combination of indirection mode and storage type. The indirection type names shown in the table are not definitive names but were arbitrarily chosen at the time of the first prototype's implementation.

The types defined in Table 3 are used to specify the types of indirect operations allowed. However, a declaration or cast must specify an entire indirection chain. In order to do this, the indirection types must be "stacked." Figure 29 shows an example C declaration and the indirection stack it produced.

The declaration in Figure 29 is declaring fp to be a pointer to a function returning a pointer to an array of 20 integers. The top element of the indirection stack specifies what indirection operations are currently allowed.

Table 3: C Indirection Types

Indirection Type	Explanation of Type
S_BASIC	No indirection. Can not be used as an address pointer. The type of the variable is its basic storage type. The bit pattern for this code's field is zero so that all variables default to this type.
S_FUNPTR	Lvalue pointer to function. Must be paired with S_LVALPTR to be legal.
S_FPTRCON	Constant pointer to function. This type describes a function definition.
S_PTRCON	Constant pointer to data. This type is given to array names.
S_LVALPTR	Lvalue pointer to data. This type describes a modifiable pointer.
S_AMPER	Pointer constant created by application of the unary & operator. This describes the result of taking the address of any lvalue.

Thus if fp were used in an expression, a unary star or subscript could be applied first and a set of function parenthesis could be applied second.

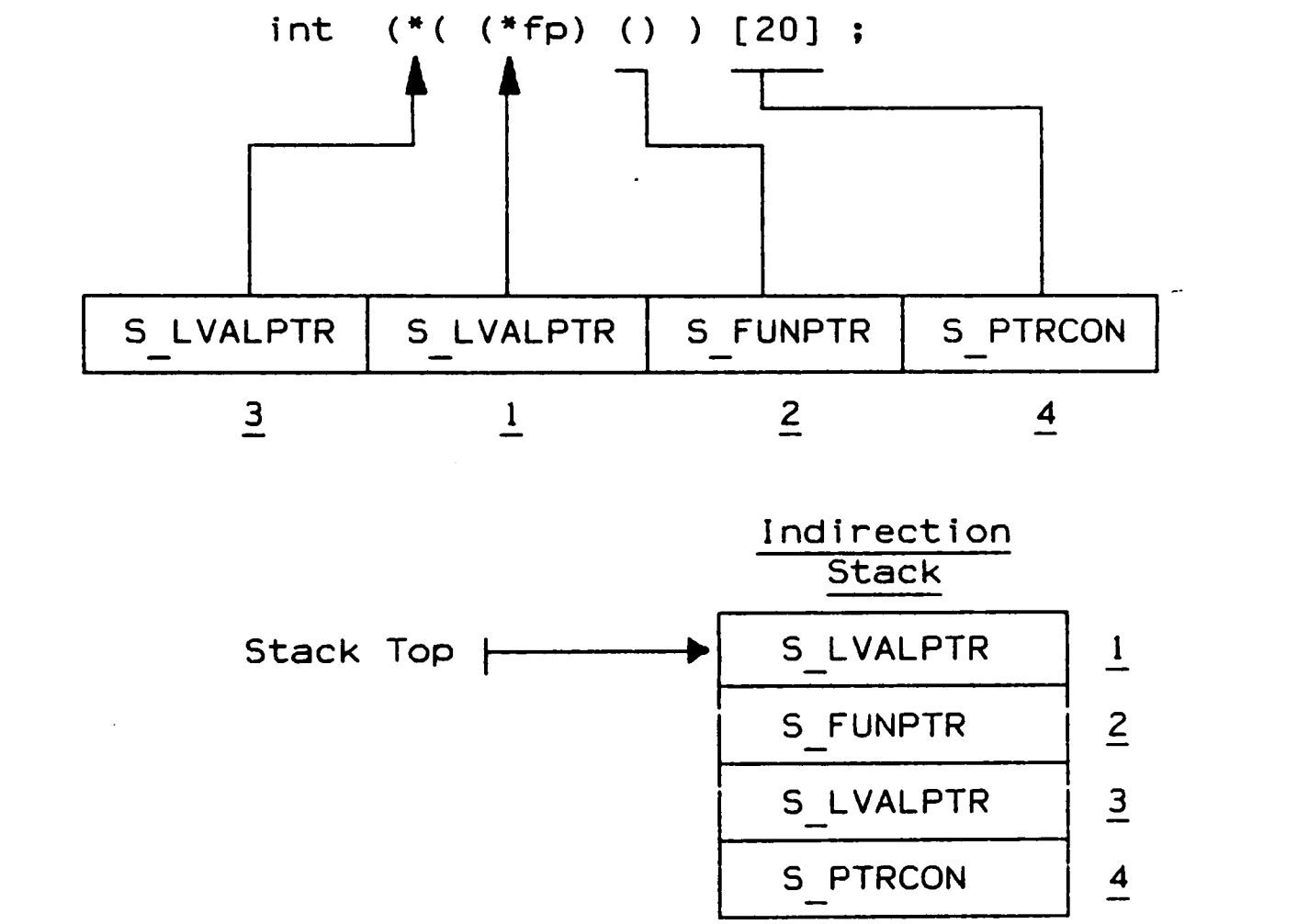


Figure 29: Indirection Stack Example

The indirection stack is linked as an attribute to the object's (fp's) symbol table node or expression tree node depending on whether the object appears in a declaration or an expression. When the object appears

in a declaration, the indirection stack is built from the indirection operators surrounding it.

When an object is used in an expression, a tree node is built for the object. This tree node contains the object's attributes copied from the object's symbol table node. These attributes include a copy of the symbol's indirection stack. If a cast operator is applied to the object in an expression, an indirection stack is built for the unary cast operator from indirection operators within the cast, and the stack is then used in the place of the object's indirection stack.

An expression tree starts out with only the leaves and cast operators having indirection stacks. The tree is traversed in postorder and modified copies of the leaves and casts' indirection stacks are placed on each operator. The rules governing the stacks' modification as they are propagated up the tree from the leaves are shown in Table 4. Once the traversal of the tree is completed, the indirection type of the result can be determined by examining the top entry of the tree root's indirection stack.

Each operator used in an expression has semantically imposed restrictions on what indirection types its operands may be. As each operator is visited during the postorder traversal, these restrictions are imposed

by examining the top few entries of each operand's indirection stack. Figure 30 contains examples of various semantic complications that can occur. In declaration 1, fp is declared to be a pointer variable containing a pointer to a function. In declaration 2, fc is declared to be a function. In statement 1, fc is treated as a constant pointer to the function and the value of the constant is assigned to the pointer variable fp.

Table 4: Indirection Stack Propagation Rules
Part 1

Legend:	μ_l(operator) = Prefix Unary Operator
	μ_r(operator) = Postfix Unary Operator
σ_l	= Left Operand's Indirection Stack
σ_r	= Right Operand's Indirection Stack
σ	= Current Operator's Indirection Stack
σ(code)	= Push code Onto Stack
$\|\sigma\|$	= Pop Stack
β(operator)	= Binary Operator
$\emptyset$	= No Operand or Empty Stack
θ	= The Set of All Operators Not Mentioned in Table.
X	= Don't Care

Table 4: Continued

Current Operator	Left Operand	Right Operand	Propagation Rule
$\mu_l(*)$	$\emptyset$	Pointer	$\sigma \Leftarrow \sigma_r$; $\|\sigma\|$
$\beta(+)$	pointer	data	$\sigma \Leftarrow \sigma_l$
$\beta(+)$	data	pointer	$\sigma \Leftarrow \sigma_r$
$\beta(-)$	pointer	data	$\sigma \Leftarrow \sigma_l$
$\beta(-)$	pointer	pointer	$\sigma \Leftarrow \emptyset$
$\mu_l(\&)$	$\emptyset$	lvalue	$\sigma \Leftarrow \emptyset$; σ(S_AMPER)
$\mu_l(++)$	$\emptyset$	X	$\sigma \Leftarrow \sigma_r$
$\mu_r(++)$	X	$\emptyset$	$\sigma \Leftarrow \sigma_l$
$\mu_l(--)$	$\emptyset$	X	$\sigma \Leftarrow \sigma_r$
$\mu_r(--)$	X	$\emptyset$	$\sigma \Leftarrow \sigma_l$
$\beta(=)$	X	X	$\sigma \Leftarrow \sigma_r$
$\beta(\theta)$	X	X	$\sigma \Leftarrow \emptyset$
$\mu_l(\theta)$	$\emptyset$	X	$\sigma \Leftarrow \emptyset$
$\mu_r(\theta)$	X	$\emptyset$	$\sigma \Leftarrow \emptyset$

After statement 1, both fc and fp may be used to call the same function. The call using fc is shown in statement 2. By symmetry, one would expect statement 3 to contain the function call using fp. But statement 3 is illegal and statement 4 contains the correct call. Thus

a semantic restriction must prevent statement 3 from being accepted. Statement 5 shows another example of a semantically prevented operation. Although fp is a pointer and applying unary star to a pointer yields the object pointed to, this operation is not allowed for function pointers. Considering the usage of fp in statement 4, it is not clear what *fp would mean. The last example, statement 6, is also illegal. J is declared to hold an integer and thus can't be used as an address.

```
int (*fp)() ;          /*declaration 1*/
int fc() ;             /*declaration 2*/
int i,j ;              /*declaration 3*/

fp = fc ;              /*statement 1*/
i = fc() ;             /*statement 2*/
i = fp() ;             /*statement 3*/
i = (*fp)() ;          /*statement 4*/
i = *fp ;              /*statement 5*/
i = *j ;               /*statement 6*/
```

Figure 30: Semantically Restricted Operations

Binary operators used to perform address arithmetic cause an even greater problem. In Figure 31, the variables ip1, ip2, fp1, and fp2 are pointer variables. The result of statement 1 is the number of integers

between the integers that ip1 and ip2 point to. Statement 2 is illegal since fp1 and fp2 are function pointers. Thus, it is not sufficient to just examine the top indirection stack entry to see if both variables are pointers. The next to the top entry must simultaneously be examined to see what is pointed to.

```
int *ip1, *ip2 ;            /*declaration 1*/
int (*fp1)(), (*fp2)() ;    /*declaration 2*/
int i, j ;                  /*declaration 3*/

i = ip1 - ip2 ;             /*statement 1*/
k = fp1 - fp2 ;             /*statement 2*/
```

Figure 31: Function Pointer's Asymmetry

The need for examining multiple indirection stack entries and the desire to implement the indirection stack in as small a storage as possible led to the indirection stack being implemented as a set of fields 3 bits in length in a 32 bit "word" (long precision in C). The least significant bit field was the top of the stack. An entry can be popped by right shifting the word by 3 bits. An entry is pushed by shifting the word left by 3 bits and oring the word with the 3 bit code for the indirection type being pushed. By proper

masking of a bit field in the word, any stack entry can be nondestructively examined.

What is not apparent from the previous algorithm is the difficulty in determining the size of an object that is being pointed to. When a pointer points to a simple type such as int, char, etc., or another pointer, the size of the pointed to object can immediately be determined via implementation defined constants. Arrays however, pose a different problem. Since C allows the reference of subarrays within a multidimensional array, the size of each subarray must be stored along with the indirection type allowing its reference. Thus, the necessary quantities which must be known at any stage in the referencing indirection chain are the size of the current object and if the current object is a pointer, the size of the object pointed to. One might conclude that the above information could be saved by simply storing the array's dimensioning subscripts.

However, the example shown in Figure 32 proves this to be inadequate. In statement 3, *array[0][0] is an lvalue pointer 2 bytes in size and it points to an array 20 bytes in size. In statement 4, **array[0][0] is an array 20 bytes in size and acts like a constant pointer to the first element of the array. This element

is a 2 byte integer. The first important point to recognize about this indirect reference chain is that *array[0] and **array[0] always represent or point to the same exact address. The only difference between the two objects is the object size and pointer type attributes that they contain.

```
int (**array[50][10])[20] ;    /*declaration 1*/

array[0] ;                     /*statement 1*/
array[0][0] ;                  /*statement 2*/
*array[0][0] ;                 /*statement 3*/
**array[0][0] ;                /*statement 4*/
(**array[0][0])[0] ;           /*statement 5*/
```

Figure 32: Scalar Separation of Array References

Since they point to the same address but have different attributes, it seems like the two forms contradict each other. Part of the confusion comes from the fact that although an array object represents an object the size of the array itself, the rules of the C language say that array objects act in an expression like a constant pointer to the first element of the array. For a one-dimensional array, this would be the first data object, and for a multidimensional array this would be the first subarray. Thus **array[0][0] represents a 10

element array whose first element is referenced in statement 5 while *array[0][0] represents a pointer variable pointing to an object the size of the array.

The second point to be recognized from the example is that array references in an indirect reference chain may be separated by one or more references to scalar objects. This means that every indirect reference in the indirection chain would need to store the size of every object being referenced. Or the most storage efficient method would be to mark an array reference and attach the array dimension to the reference. It can be implemented by specifying that all pointer constants have a size associated with them and all other pointer types have a size of two bytes. This would cover all array references since all array objects act as pointer constants. To minimize implementation dependent constants, the size is always stored in units of the storage type housed in the array. The sizes are stored in a size stack in the same order that their corresponding constant pointer indirection type (S_PTRCON) entries are stored in the indirection stack. When the indirection type S_PTRCON is popped from the indirection stack, a size is popped from the size stack. Each size stack entry is 16 bits (2 bytes) in precision. The indirection and size stacks that would be created by

declaration 1 in Figure 32, and the stacks' relationship are shown in Figure 33.

Figure 33 shows that this implementation allows convenient size determination. By examining the next to the top entry's indirection type, the size of the object being pointed to can be determined. If the next to the top entry's type is not S_PTRCON then the size is determined via implementation dependent constants. Otherwise, the size stack is consulted. It is important to find the size of objects pointed to because operators such as ++ make use of this information.

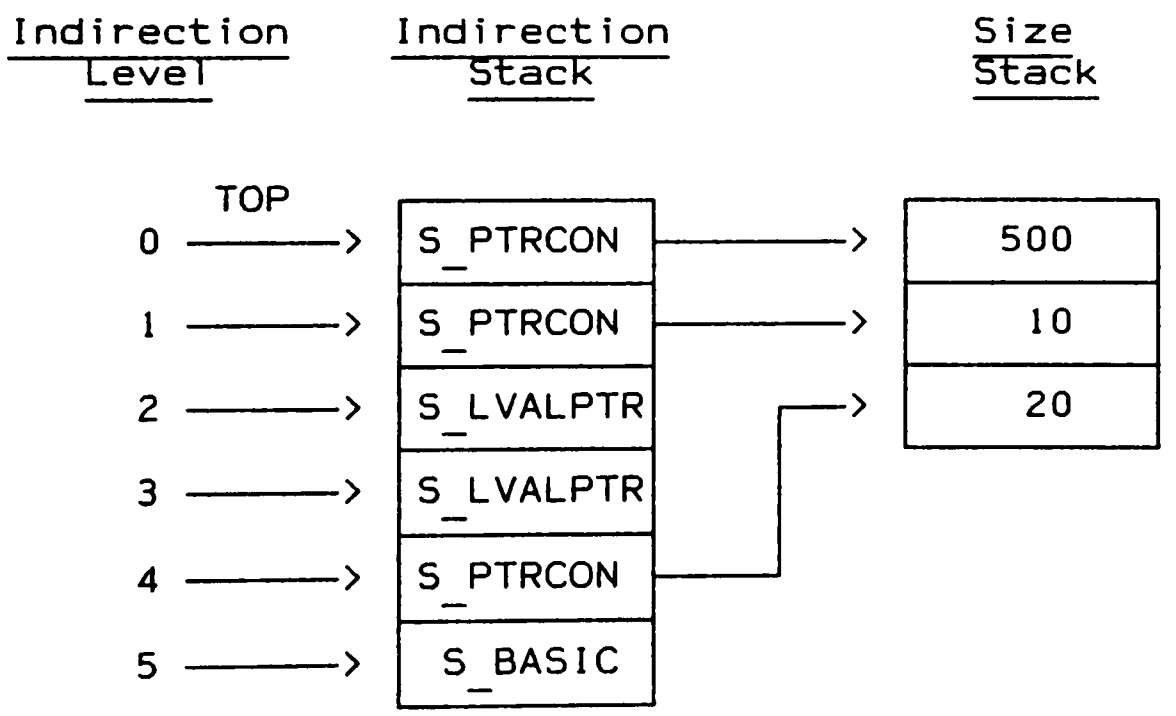


Figure 33: Implementation of Size Stack

To allow the implementation of indirection in expression trees to be as easy as possible, extensive decoding of indirection in declarations and casts is necessary. The same algorithm can be used to decode the indirection in both casts and declarations. Since the algorithm is totally modular with well defined pathways entering it and leaving it, it will be considered as an entity separate from the parser. At the time the indirection algorithm receives control, the context and storage type information surrounding the indirection to be decoded has been analyzed to the fullest extent possible. The parser communicates contextual information to the algorithm via a selected subset of its status flags. The parser flags used by the algorithm are shown in Table 5.

The first flag of importance is the Γ(TP_TYPEACT) flag, since it affects the algorithm's initialization. User defined type names contain indirection as well as storage precision information. This initial indirection must be attached to the variable declared. Any indirection operators surrounding the variable must build upon this initial indirection.

The declaration indirection algorithm is recursive, with each level of the recursion representing an indirection level. The algorithm decodes a single

variable's indirection as it recurses down to the lowest level. It then builds an indirection stack for the variable as the recursion unwinds back to the initial level.

Table 5: Parser Flags Used in Indirection Translation

Flag Name	Explanation of True State's Meaning
Γ(TP_TYPEDEF)	The typedef keyword has been recognized. A type definition is in progress.
Γ(TP_TYPEACT)	Type action. A defined type name is being used to type the identifier being declared.
Γ(TP_CAST)	A start cast construct has been recognized. The resulting indirection information should be attached to a cast operator. No symbol table node is to be created.
Γ(TP_MEMBER)	The current declaration is declaring a member to a structure.

Note: Γ is the shared context flag word discussed in the lexical analyzer section. The flags listed in the table reside in this flag word.

It is important for the algorithm to know when it is at the focus of the declaration, which is at the lowest level in the recursion. The focus in a declaration is the identifier being declared. In a cast, the focus is the point at which the identifier would appear if the cast was a declaration instead. This position is the point at which a right hand indirection operator (subscript bracket pair or function parenthesis) or a right parenthesis is first encountered. The left parenthesis operator adds a further complication to the recognition of a right hand operator since it can occur as either a left or right hand operator. If a right parenthesis immediately follows it, then it is a right hand operator (function call parenthesis). Otherwise, it is a left hand operator. The kind of focus searched for is determined by the truth of the ⌜(TP_CAST) flag.

Before the indirection decoding algorithm can be described, some terminology must be reviewed and further explained. The terminology explained in the points below and illustrated in Figure 34.

1. An indirection region is a section of the declaration which is enclosed by a nonredundant pair of parenthesis. This section excludes the part enclosed by a inner indirection region.

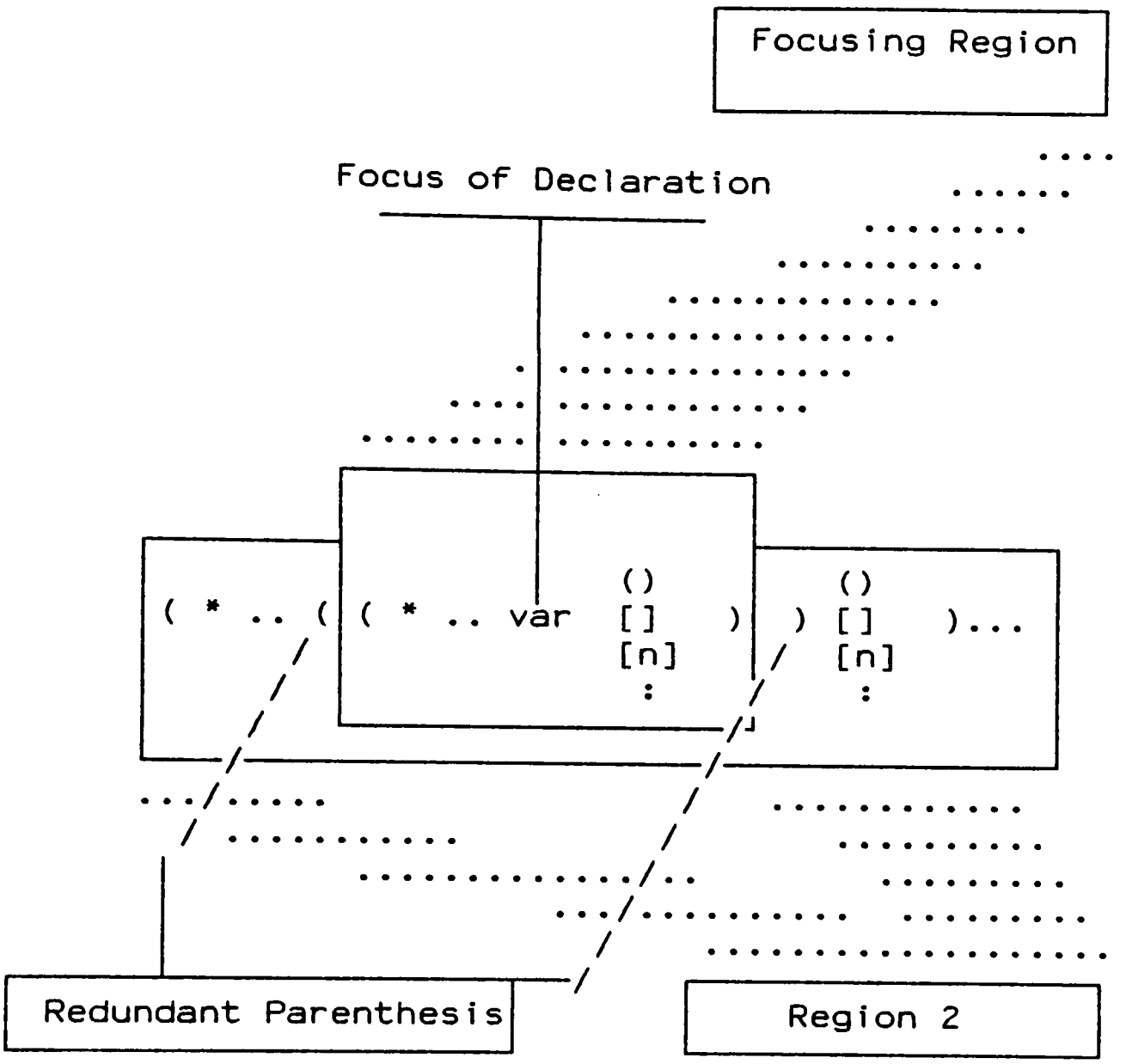


Figure 34: Definition of Indirection Region

2. A nonredundant pair of parenthesis is a pair of parenthesis which enclose a new set of unary stars not enclosed by a more inner pair of parenthesis.

To visualize the bounds of an indirection region, look at Figure 34. In the figure, the focusing region is the innermost indirection region. Region 2 is the next indirection outward. As shown in the figure, region 2 does not include the focusing region. Now that the terms have been defined, the indirection decoding algorithm can be summarized by the points below:

1. Initialize the indirection stack (Current Indirection Word, CIW) and find the declaration focus.
2. For each indirection region starting from the focusing region and working outwards do the steps outlined below. For each indirection region, start with the CIW resulting from the previous indirection region.
   A. Make a left to right scan of the part of the indirection region to the right of the focus. For each right hand indirection operator, shift in its 3 bit type code into the CIW.
   B. Make a right to left scan of the part of the indirection region to the left of the focus. For each unary star encountered, shift in the 3 bit code S_LVALPTR into the CIW.

There are quite a number of restrictions not reflected in this simplified algorithm. A few of them are listed below:

1. If Γ(TP_MEMBER) is set (member declaration) then the function parenthesis right hand indirection operator, (), may not be used in the focus region, i.e., a member may not be a function.
2. If Γ(TP_CAST) is set (decoding a cast), then an identifier may not appear at the focus.
3. The function parenthesis and subscript bracket pair, [], may not appear in the same region.

This completes the description of the implementation of C indirection. This algorithm has only been tested on paper.

Expression Analyzer: Design and Implementation

Overview:

The C compiler expression analyzer decodes C expressions, performs a legality check of the C expression at the semantic level, and rearranges the C expression for optimum code generation. It accomplishes these tasks using the following steps:

1. An expression tree is constructed from the parsed input.
   A. The expression tree is constructed using a modified version of the standard bottom up priority driven construction algorithm.
   B. The expression context commands required by the lexical analyzer are generated. These commands were previously discussed in the lexical analyzer section.
2. The expression tree is checked for semantic violations.
   A. Illegally typed operands.
   B. Illegal use of indirection, i.e., call of a nonfunction, use of a nonaddress as a pointer, etc.
3. The expression tree is rearranged for code generation.
   A. The tree is linearized (made left heavy) via transformations.
   B. Types are propagated up from the leaves to the root. This propagation consists of the following steps:
      a.) The type of an operator's result is determined based on the types of

the operands. The C arithmetic conversion rules discussed at the beginning of the chapter are used to determine the result's type.

b.) A unary conversion operator is inserted between the operator and any operand not matching the operator's type.

c.) Subscripts and implied subscripts (numbers added or subtracted from a pointer) are replaced by a subtree containing an expression which converts the subscripts into offsets.

C. Indirection is propagated from the leaves to the tree root.

D. The constant folding algorithm collects and precalculates the constant parts of the expression. This not only removes constant expressions generated by the user but constant expressions injected by the compiler. These injected constant expressions include those produced by pointer and subscript conversions.

E. The interchange linearization algorithm is applied to the tree.

F. Unary stars are replaced by indirect reference or indirect store depending on their context.

4. A postorder traversal is performed on a correct tree. As each node is visited, the code generator is called with its contents.

The checks performed in step 2 actually occur during the transformations discussed in step 3. Some of the more difficult of these transformations are discussed in the following subsections.

Unary Star Conversion

Unary star operators in C can either represent indirect storage or indirect reference operators depending on whether they occur on the left hand or right hand side of an assignment. Since usage context is totally unknown by the code generator, the unary star's context dependency must be removed by replacing it with an indirect store or indirect reference operator. The difficulties in doing these substitutions can be summarized as follows:

1. C allows multiple and nested assignments to occur in one expression statement, which makes it difficult to determine whether the

unary star is on the reference or assignment side of an assignment operator.

2. The reference/storage must be classified according to class of the operand:

   A. Automatic Symbolic Constant Address. The address is a symbolic constant, i.e., an array name and represents an offset on the frame pointer. If the subscript is a numeric constant, it is also incorporated into the operator, since this address expression can be directly expressed as an assembly code operand.

   B. Static Symbolic Constant Address. This is the same as the previous classification except that the result is an absolute address and not an offset.

   C. Constant Absolute Address. The address is a numeric constant and is an absolute address.

   D. Register Indirect Reference/Storage. The operand of the unary star is an lvalue. It can't be combined with the operator. The address must be fetched from the lvalue operand into a register. A

register indirect addressing mode is then used to access the target location.

Multiple and nested assignments are a problem because it is difficult to determine the reference and storage side of each assignment operator in the tree. The multiple assignment in statement 1 in Figure 35 would lead one to believe this is an easy task. However, when multiple assignment is combined with nested assignments as shown in statement 3, determination of context becomes difficult.

```
A = *B = C + D ;                    /*Statement 1*/
A = *(B + *E) = C + D ;             /*Statement 2*/
A = *(B + (*F = *E)) = C + D ;      /*Statement 3*/
```

Figure 35: Assignment Context Determination

Statement 2 shows that indirect referencing can occur on the storage side of an assignment operator. The value pointed to by E is referenced and added to B to form the destination address for the assignment. Statement 3 assigns the value pointed to by E to the

location pointed to by F prior to adding the value to B.

A search of more complex versions of these statements for generalizations leading to an algorithm was fruitless. However, when the expression trees for the same statements were analyzed, the key generalization was revealed. In each expression tree a binary indirect store operator can be synthesized by combining the assignment operator with its left hand unary star operand. The algorithm is now apparent and can be stated as follows:

1. Perform a post order traversal of the expression tree.
2. If the node visited is an assignment operator with a unary star left operand, convert both nodes into a single binary indirect store operator and perform the following:
   A. Perform a post order traversal on the left subtree after the conversion.
   B. Convert each unary star operator node into an indirect reference operator node.

Because of C's numerous assignment operators (+=, *=, etc.) and the different classes of storage previously discussed, quite a number of assignment operators

could be produced. In order to half the number of cases, each assignment statement is placed in a canonical form. All assignment operators are converted to indirect store operators and if the left operand is not a unary star, the address of the left operand is taken. By canonizing assignment statements, the following optimizations are automatically performed:

1. Array assignments are automatically performed using the correct mode of addressing: automatic/static symbolic indirect, or register indirect.
2. Automatic variables are accessed properly via frame pointer offsets.

Tree Linearization

The purpose of tree linearization is to reduce the number of stacked values during the evaluation of an expression. In a standard expression tree, if the right operand of a binary operator is an expression, then that expression will have to be evaluated before the current binary operation is performed. If there are an insufficient number of registers to evaluate the right subtree, then the left operand must be temporarily pushed onto the stack (dumped) so that the register containing the left operand may be freed for use. The

operand is restored after the right subtree has been evaluated. This register dumping potentially has to be performed at every binary operator node containing more that one node in its right subtree. By reducing the depth of the right subtree, the number of register dumpings can be reduced. Figure 36 show the linearizing transformations applied to the expression tree.

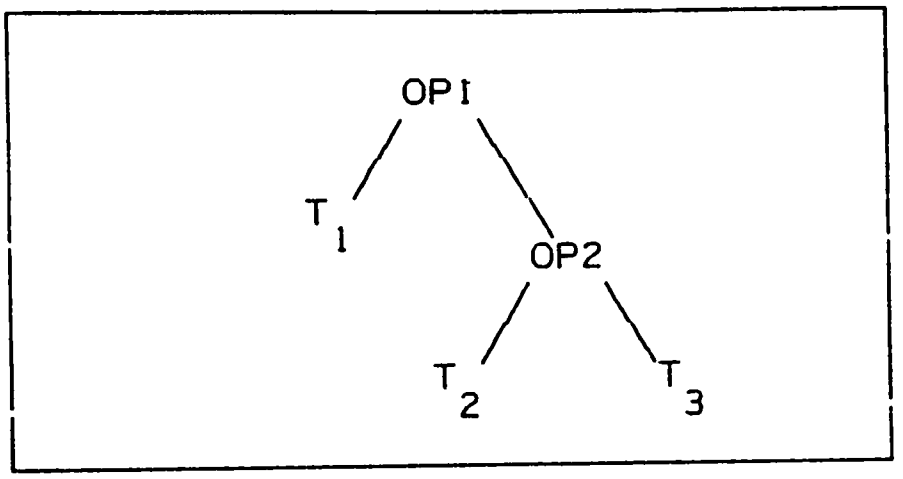
A linearization transformation is applied when a section of the expression tree matches the transformation template. The template is satisfied by simultaneously matching the pattern tree shown in the figure and having the two operators be a member of the set of operators for the transformation. The set for each transformation is shown in the figure above the input pattern for the transformation.

In addition to applying the linearization transformation, the tree can be made heavier on the left side by locating binary operators which are commutative or reversible and comparing the depth of the left and right subtrees. If the right subtree is deeper, then the right and left subtrees are interchanged (interchange transformation).

Legend:	T_1, T_2, T_3	=	Subtrees
	OP1, OP2	=	Binary Operators

Transformation #1 Input

(OP1,OP2) ϵ (*,*),(/,*),(+,+),(-,+),(&,&),(¦,¦),(^,^)



Transformation #1 Output

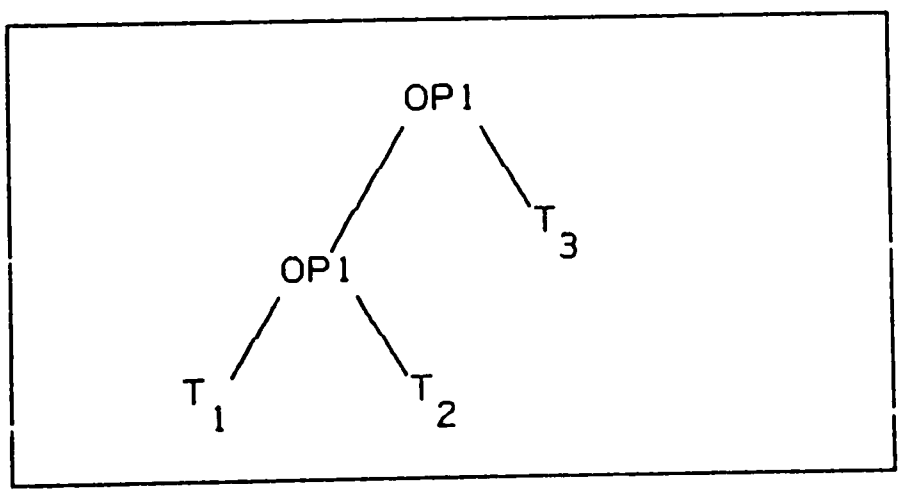
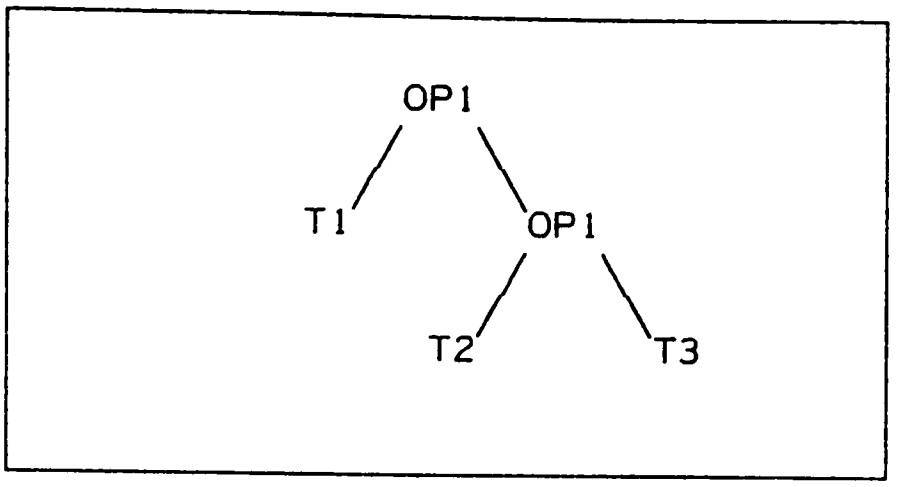


Figure 36: Tree Linearization Transformation
Part 1

Transformation #2 Input

(OP1,OP2) ∈ (-,+),(/,*)



Transformation #2 Output

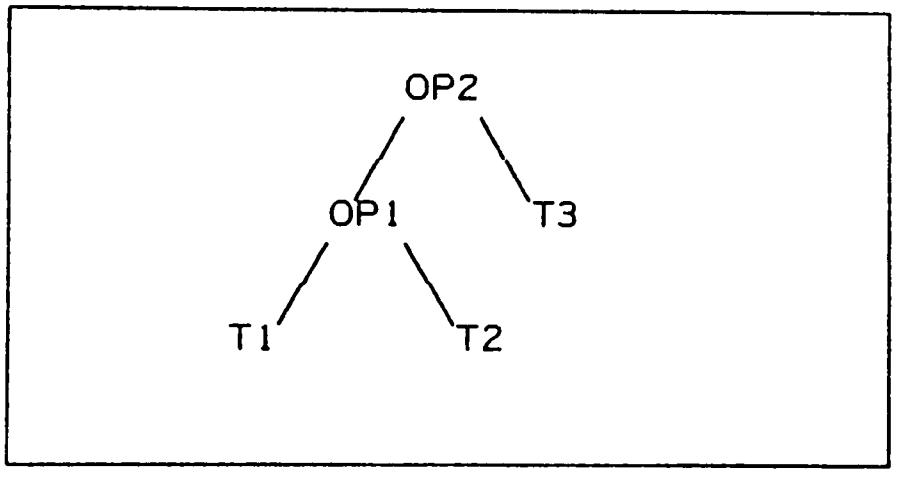
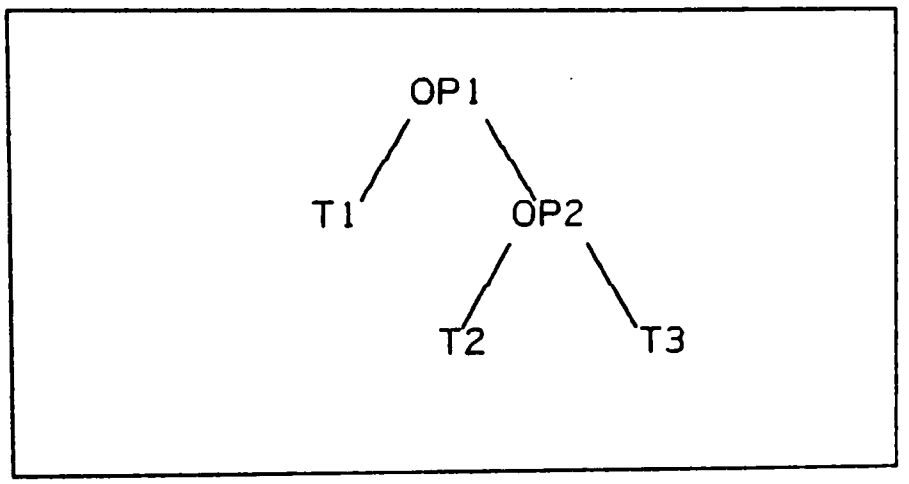


Figure 36: Tree Linearization Transformation
Part 2

Transformation #3 Input

(OP1,OP2) ∈ (+,-),(*,/)



Transformation #3 Output

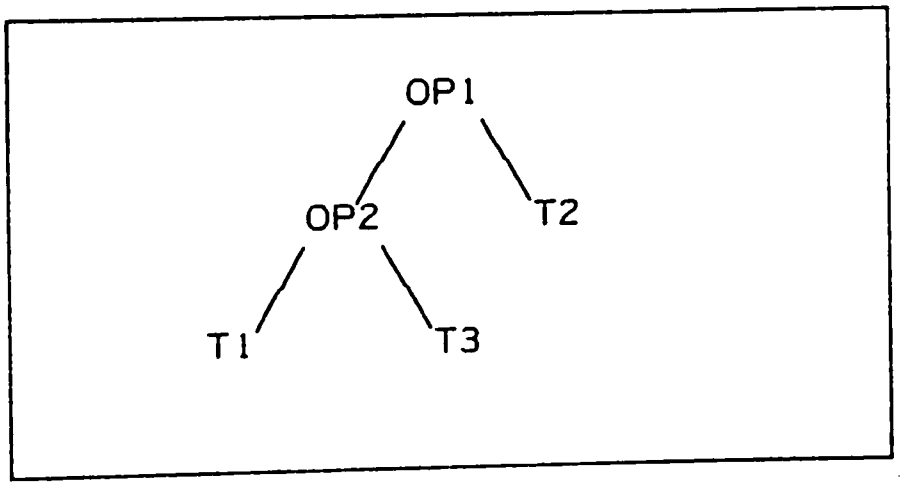


Figure 36: Tree Linearization Transformation
Part 3

Reversible operators are different from commutative operators in that they are not commutative. A reversible operator is an operator for which another operator performing the same exact operation exists. This second operator differs from the first, though, in that its left and right operands are interchanged. Thus, when the operands of a reversible operator are interchanged, the operator is also changed to its reversed counterpart. Subtraction is an example of a reversible operator ($-,R^{-}$). Now that the transformations have been defined the linearization algorithm can be stated as follows:

1. Traverse the entire expression tree applying the linearization transformation where the restrictions and conditions are satisfied.
2. Repeat step 1 until no changes take place in the tree.
3. Perform other optimization algorithms.
4. Traverse the entire expression tree and apply the interchange transformation where appropriate.

Constant Folding

The purpose of constant folding is to locate and evaluate subexpressions whose value can be calculated

at compile time. C requires constant folding due to the constant expressions injected by the compiler's conversions and due to the coding style of C programmers. C compilers implement "named constants" via a preprocessor pass. Thus, rather than the programmer hand calculating the constant expressions at the time he writes the code, C programmers generally place the "names" for the individual constants into an expression statement. This results in constants being scattered throughout the expression statement and requires that the compiler collect them into a constant subexpression which can be evaluated at compile time.

Some compilers such as TI Pascal and other multi-pass TI compilers based on the SILT intermediate language go to extensive effort to simplify expressions involving constants[20]. Such efforts include rewriting expressions so that more time efficient operators are used in the expression.

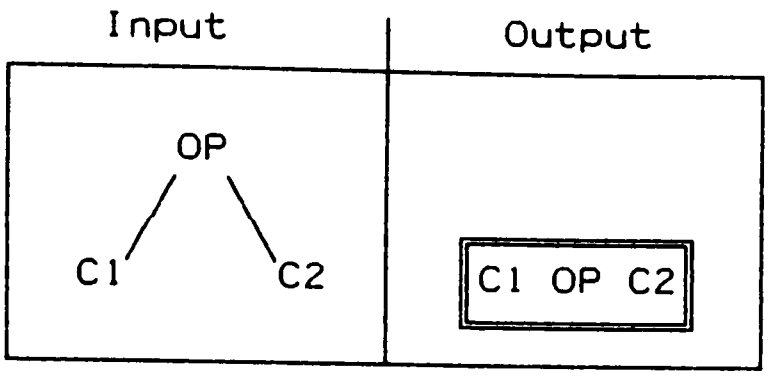
For the TMS7000 C compiler, the goals are to not overly obscure the flow of the original source code and to provide a fast simple compiler. This means that constant folding optimizations are restricted to rearranging the expression's order of evaluation so that constants are collected into one single subexpression which can be evaluated at compile time.

a canonical form. This means that the primary task of the constant folding algorithm is to collect constants by moving up from the leaves of the tree towards the tree root. As purely constant subexpressions form, they are reduced to a single constant.

To accomplish this task, the expression is rearranged using the associative and distributive laws. The first step is to apply the group 1 transformations, shown in Figure 37, which evaluate already existing constant subexpressions and prepares the expression for the associative and distributive transformations.

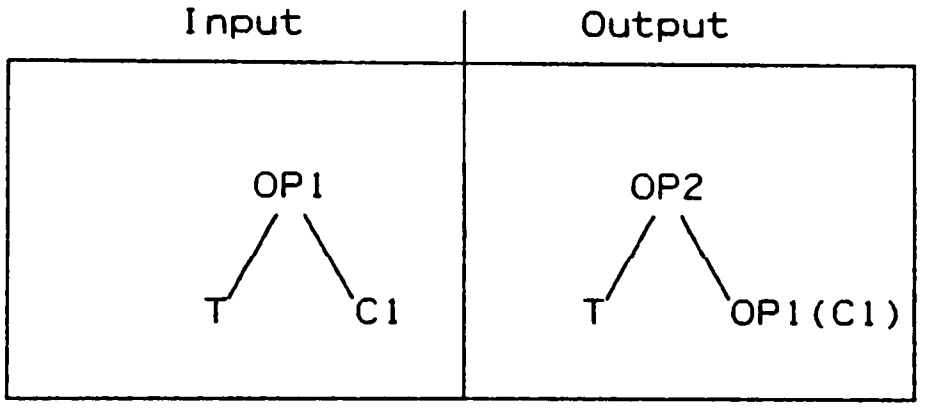
Transformation 2 gives the constant greater mobility in the tree by converting the operator to a commutative operator. The OP2(C1) in transformation 2 designates that the constant has been marked with a delayed operation. In the case of (OP1,OP2) ⇒ (-,+), this mark means the constant is negated. In the case of (OP1,OP2) ⇒ (/,*), this mark means that the true value of the constant is 1/C1. The constant is not immediately inverted since only integer arithmetic is used (A problem unique to a no floating point compiler). Rather, it is divided into the constant it is eventually combined with.

Transformation #1



Transformation #2

(OP1,OP2) ∈ (-,+),(/,*)



Legend: C1, C2 = Constants
T = Subtree

Figure 37: Group 1 Transformations

The next group of transformations, shown in Figure 38, apply the associative and distributive properties to the expression in an effort to move the constants up the tree.

In Figure 38, the dotted (...) tree links represent any number of intervening operator nodes. The only restriction on these intervening operators is that they must be the same operator as either OP1 or OP2. Cx and Tx in the figure represent constants and subtrees respectively. The list of ordered pairs above each transformation contains the set of all operators for which the transformation is valid.

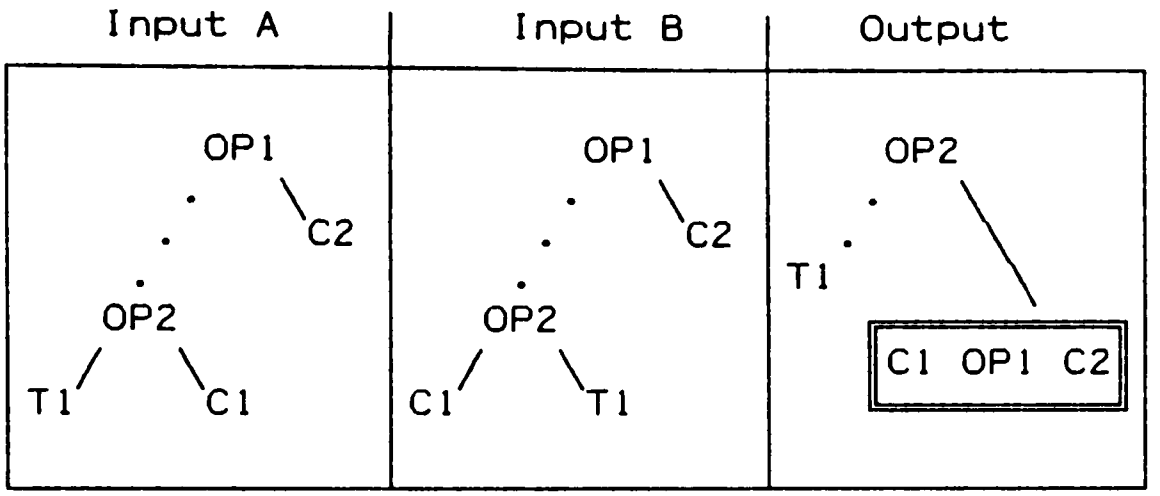
The last transformation group, shown in Figure 39, uses the distributive law to group associative operator sets together. This potentially allows group 2 transformations to be further applied to the tree to achieve greater simplification.

The constant folding algorithm can be described as follows:

1. Traverse the tree in postorder and apply the group 1 transformations.
2. Traverse the tree in postorder and apply the group 2 transformations.
3. Traverse the tree in post order and apply the group 3 transformation.

Transformation #1

(OP1,OP2) ∈ (*,*),(/,*),(+,-),(+,+),(&,&),(|,|),(^,^),
(&&,&&),(||,||)



Transformation #2

(OP1,OP2) ∈ (*,*),(/,*),(+,-),(+,+),(&,&),(|,|),(^,^),
(&&,&&),(||,||)

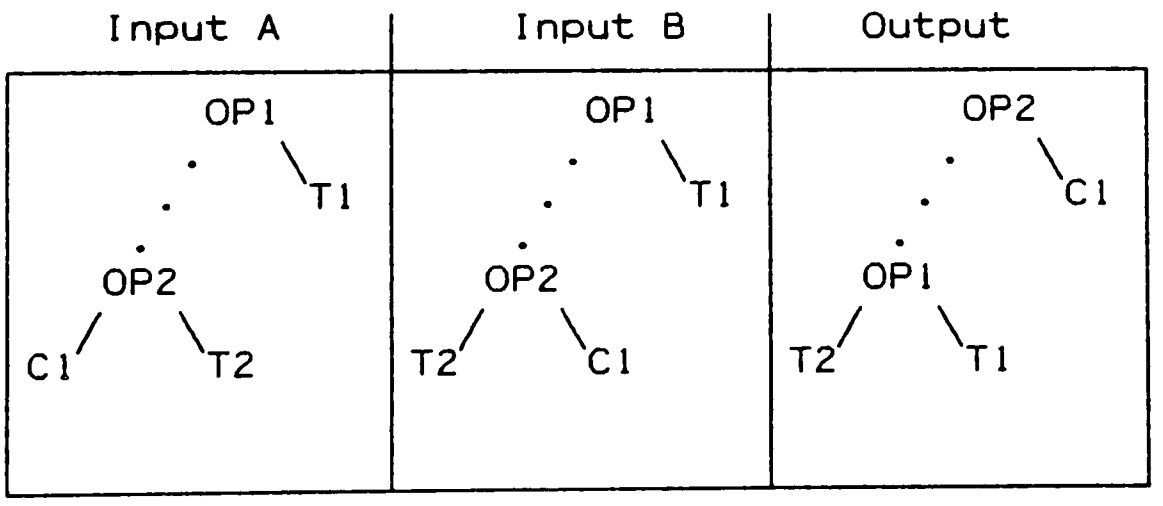
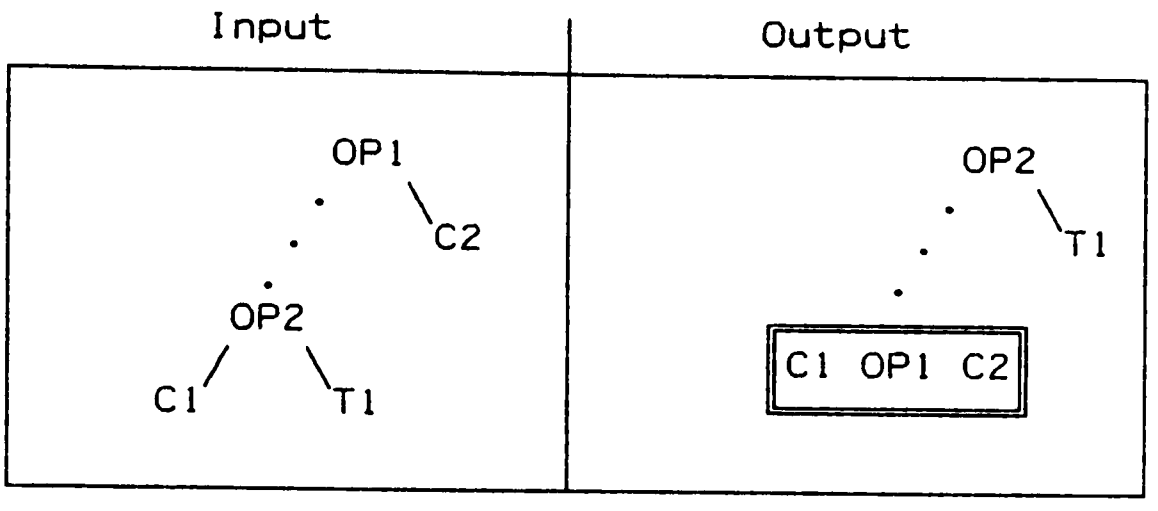


Figure 38: Group 2 Transformations
Part 1

Transformation #3

(OP1,OP2) ∈ (+,-),(*,/)



Transformation #4

(OP1,OP2) ∈ (+,-),(*,/)

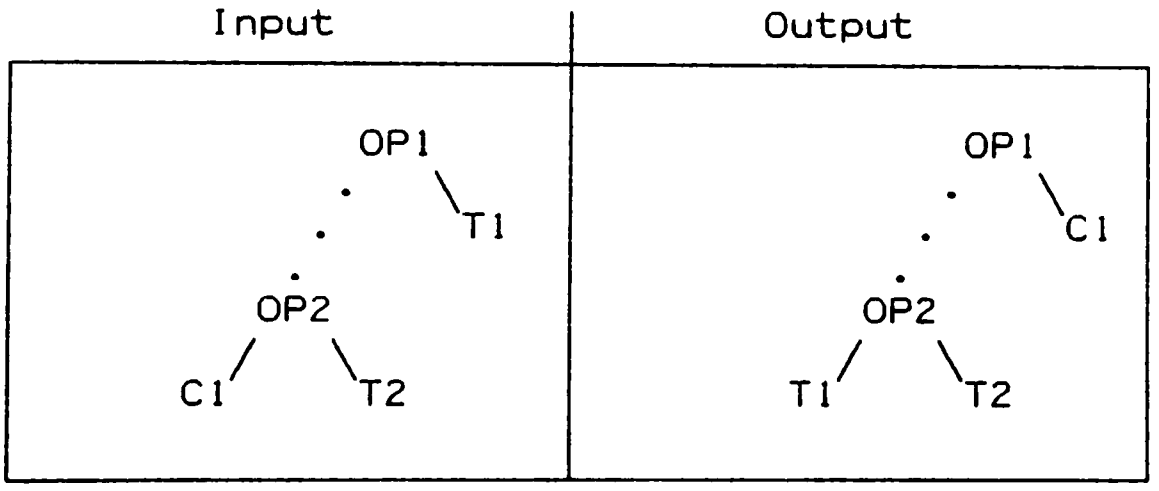


Figure 38: Group 2 Transformations
Part 2

4. Repeat steps 1, 2, and 3 until no changes are made.

This completes the discussion of the implementation of the expression analyzer. The design and implementation was never completed due to the problems outlined in the next chapter.

Transformation #1

(OP1,OP2,OP3) ∈ (*,+,+),(*,+,-),(*,-,+),(*,-,-),
(/,+,+),(/,+,-),(/,-,+),(/,-,-)

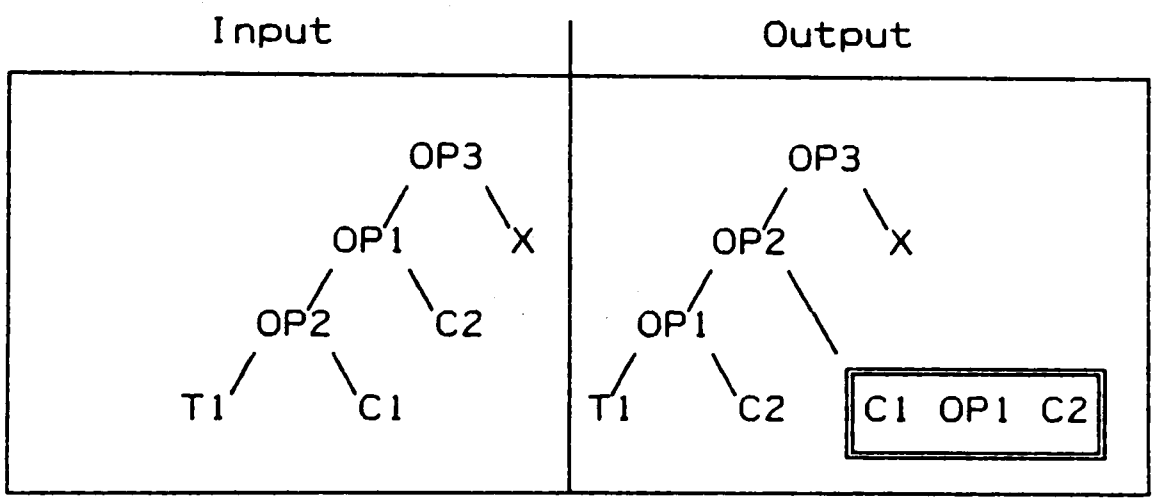


Figure 39: Group 3 Transformation

CHAPTER 6

PROBLEMS WITH THE IMPLEMENTATION AND DESIGN

Overview

One of the major problems with a fundamentally new project, such as the TMS7000 C compiler project, is that major design decisions must be made before the actual detailed knowledge required to do the project has been obtained. In fact, the relationship between knowledge, design decisions, and experience is circular. In order to make a design decision, detailed knowledge must be available. Since hardware intimate compilers are not well known, there was limited previous knowledge available. Thus, in order to have detailed knowledge, implementation of this new compiler must be experienced by the potential implementors. In order to implement the compiler, a design must be present. Although there are many C implementations, there are no implementations providing low level access to the microprocessor hardware. The goal of most C implementations is to maximize hardware independence.

Most modern implementations of C are based on bottom up (YACC generated) compilers. Only the original C compiler, which implemented a subset of the current language, is top-down. Thus the design decisions for

the top-down implementation of the TMS7000 C compiler were based on the union of requirements imposed by microprocessor programmers and experience with bottom up hardware independent compilers. As a result, the design did not produce an implementation which could be viably completed in a limited time scale. The implementation was necessary, however, to obtain the experience required for a more successful design.

The implementation problems discussed in the next chapter are not truly fatal in the sense that it would be impossible to produce a compiler. The problems should be viewed as having introduced so many complexities that the implementation can not be completed in a reasonable amount of time. Redesign of the compiler would produce a better utilization of time. The problems with the compiler's implementation can be summarized as follows:

1. Failure of the original design to provide all of the necessary hardware access routes required by assembly language programmers.
2. Register access method's interference with the expression tree algorithms.

The first problem occurred because the original design was forbidden from including language changes. Had language changes been implemented initially, they

would have not been as focused towards solving the hardware access problem. The problem of implementing hardware access became more sharply defined during this first attempt at the compiler.

The remaining problem was discovered during the implementation of the expression analyzer. It will be discussed in detail in the next section.

Register Access Problems

The first problem to surface during implementation of the expression analyzer was the interference of the register access method with the unary star conversion algorithm. A register access is recognized by the combination of a unary star and a constant in the correct value range containing indirection. When a register access appears on the left side of an assignment, the unary star is stripped from it since the algorithm requires the left side of an assignment to be a destination address. The problem is that the register "address" must be marked so that it is treated differently from a normal constant address. What is normally generated for a constant address is the following:

1. Move immediate constant to register.

2. Store indirect the contents of the result register using the register containing the constant as the address.

If the constant represents a register access, it should not be loaded into a register since it is the register number itself. During the implementation, this situation was declared to be a special case and was handled by creating a register store operator. The fact that this situation was a special case has the following significant implications:

1. The attribute "register" is given by the compiler and is not related to any declaration or explicit user typing. Instead the attribute must be recognized from the context of the operand's usage.
2. No blanket rule can exist for converting a register access in every situation. This is the result of context recognition verses explicit typing.
3. Every context in which a register access can occur must have a special case code block in the compiler to handle it.

Especially troublesome operators are postincrement, postdecrement, preincrement, predecrement, and all of the arithmetic assignment operators because they

involve both an indirect reference and store to the same location. These operators are normally made efficient in a standard C compiler by caching the address of the location being operated on in a register. For a register operand, this can't be done.

Another set of special cases is generated by the statements in Figure 40. In statement 1, the register's address is added to a variable and then referenced. In this case, register access is impossible without self modifying code. Because the compiler can't know whether the compiled code will be in RAM or ROM, it must access the resulting address as a standard memory location. However, in statement 2 register access may be used since the subtracted operand is a constant.

```
char i ;
*(((char *)35) - i) = 10 ;     /*Statement 1*/
*(((char *)35) - 5) = 10 ;     /*Statement 2*/
```

Figure 40: Register/Memory Access Special Case

The net result of the previously mentioned cases is that a large percentage of the expression analyzer

becomes devoted to these special cases. Because of the structure of the compiler, much of this special case code is redundant. Cases which are not special or can be categorized can be collected into a single subroutine. Thus, the categorized or type driven cases are far more compact.

Another bad side effect of the special case approach is that closure (coverage of all possible special cases) can't be established. This means that relatively stable code for the expression analyzer can be disrupted at any time with the discovery of another special case to be implemented.

Closure is necessary in order for the compiler to have a predictable behavior from the user's point of view. If very few of the special cases are implemented, then the TMS7000 C compiler would be no different than a standard C compiler. The compiler would have failed to meet its design goals.

The solution to the problem is to introduce a type to categorize register and port access. This categorization would remove the majority of special cases and reduce the complexity of the expression analyzer.

CHAPTER 7

Conclusion

The TMS7000 C compiler project accomplished the following tasks:

1. It identified the nature of a control microprocessor via the description of the TMS7000 microprocessor and the comparison of it with other low level microprocessor architectures.
2. It demonstrated the suitability of C for control microprocessors and other hardware critical applications.
3. Assembly language features important to control microprocessor programmers were identified.
4. An improved algorithm for top-down parsing, translation, and correctness verification of C structure declarations was discovered.
5. An algorithm for top-down translation of C indirection was demonstrated.
6. A set of C expression optimizations suitable for control applications was shown.
7. A set of general microprocessor independent C language extensions required by control and

other hardware critical applications was specified.

Because of the fundamentally new approach of intimate hardware access from a high level language and the lack of generally available top-down parsing algorithms specifically for C, the development of the TMS7000 C compiler was necessarily a two pronged approach.

1. Practical problems associated with the implementation of C compilers had to be dealt with. This included the development of the following algorithms:

   A. The RAT algorithm for top-down translation of structure declarations. As demonstrated in the thesis, the translation of structure declarations is a problem not well addressed by many existing and well known compilers.

   B. A top-down algorithm for translating C indirection had to be developed. In bottom-up compilers, this translation is handled largely by the YACC parser generator. Because of the scarcity of top-down implementations of C, this algorithm is not widely available.

C. An effective method of storage management for the compiler was developed. Storage management was a crucial issue because of the memory limitations of the 8086 based PC upon which the cross compiler was implemented.

2. The practical problem of achieving the intimacy of hardware access required by control programmers was addressed. The difficulty in achieving the required hardware access from within the current C language was demonstrated. The difficulties discussed centered on the interference of hardware access techniques with the expression optimization algorithms of the compiler.

The net result of the two pronged approach was the advancement of publicly known knowledge on top-down C compiler technology and the definition of the problems associated with intimate hardware access from a high level language. The interests of control microprocessor programmers have been served.

REFERENCES

1. TMS7000 Assembly Language Programmer's Guide, Texas Instruments Inc., Houston, TX, 1983.

2. TMS7000 Family Data Manual, Texas Instruments Inc., Houston, TX, 1983.

3. A. Osborne, An Introduction to Microcomputers, Adam Osborne and Associates Inc., Berkeley, CA, 1977.

4. D. A. Patterson and R. S. Piepho, "Assessing RISCs in High-level Language Support," IEEE Micro, Vol. 2, No. 4, Nov. 1982, pp. 9-19.

5. Talking Alarm Clock/Controller Program, Internal Program Listing, Texas Instruments Inc., 1984.

6. S. C. Johnson and B. W. Kernighan, "The C Language and Model for Systems Programming," Byte, Vol. 8, No. 8, Aug 1983, pp. 48-60.

7. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language," The Bell System Technical Journal, Vol. 57, No. 6 Part 2, Jul.-Aug. 1978, pp. 1991-2019.

8. B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall Inc., Englewood Cliffs, NJ, 1978.

9. S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the Unix System," The Bell System Technical Journal, Vol. 57, No. 6 Part 2, Jul.-Aug. 1978, pp. 2021-2036.

10. A Tour Through the Portable C Compiler, Technical Report, Bell Laboratories, Murray Hill, NJ. .

11. J. E. Hendrix, The Small C Handbook, Reston Publishing Company, Reston, Virginia, 1984.

12. Unix Version Six Operating System, Internal Program Listing, Bell Laboratories, 1975.

13. PL/M-86 Users Guide, Intel Corporation, 1982.

14. Chris Moller, "Texas Instruments Microcomputers." In Lister,Paul F. (ed.), Single-chip Microcomputers, McGraw-Hill Book Company, New York, N.Y., 1984.

15. Brad Taylor,"Symbol Table Detailed Design," Texas Instruments 990 C Design Notes, Internal Document, Texas Instruments Inc., Austin, TX, 1982.

16. Knuth, D.E., The Art of Computer Programming, Addison-Wesley Publishing Company, Reading, Massachusetts, Vol. 1, pg. 423-432.

17. Standish, Thomas A., Data Structure Techniques, Addison-Wesley Publishing Company, Reading, Massachusetts, pg. 30-31, 233-235.

18. Robert M. Graham, "Bounded Context Translation." In Saul Rosen (ed.), Programming Systems and Languages, McGraw-Hill Book Company, New York, NY, 1967, pg. 184-203.

19. Nell Dale and David Orshalick, Introduction to PASCAL and Structured Design, D. C. Heath and Company, Lexington, Massachusetts, 1983 pg. A6.

20. TI Pascal Compiler (SILT) Design Document, Internal Document, Digital Systems Group, Texas Instruments Inc..

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Disagree (Permission not granted)	Agree (Permission granted)
_____	_____
Student's signature	Student's signature
_____	_____
Date	Date / /