The Second Pass of the Portable C Compiler

John Lions

Bell Laboratories
Murray Hill, New Jersey

June 1979

Preface

This document attempts a detailed examination of the source code for the second pass of the Portable C compiler. It is intended for persons with an active interest in transferring the compiler to new machines, for persons interested in maintaining and refining existing versions of the compiler, and for persons who are merely curious about the details of one interesting and fairly general approach to the problem of code generation.

The Portable C compiler is a compiler for the C language that was written by Stephen C. Johnson. It is intended to be more easily transferable to new machines than was the original compiler for the PDP11 written by Dennis M. Ritchie. The first working version of the Portable C compiler was for the Interdata 8/32; it was used to demonstrate the portability of the UNIX operating system from the PDP11 to a machine that was not under consideration when UNIX was designed. Since that time, the Portable C compiler has been transferred successfully to several other machines, so that the list of versions of the Portable C Compiler (as of April 1979) includes:

Data General Nova
Honeywell 6000
IBM System /360 and /370
Intel 8086
Interdata 8/32
PDP11
Tandem 16
VAX11/780

Not all these different versions were adapted by Steve Johnson from the original. The PDP11 version that is the principal subject of this document was adapted from the Interdata 8/32 version by H. Lee Benoy.

The functioning of the compiler as a whole is described in "A Tour through the Portable C Compiler" by Stephen C. Johnson, in the *UNIX Programmer's Manual, Seventh Edition*, Volume Two. Other references relating to the present work are "Portability of C Programs and the UNIX System" by S. C. Johnson and D. M. Ritchie, *Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 2021-48, July-August 1978, and "A Portable Compiler—Theory and Practice" by S. C. Johnson, *Proc. Fifth ACM Symposium on Principles of Programming Languages*, January 1978.

I would like to acknowledge the considerable assistance of several people who reviewed drafts of this document. Particular among these have been Steve Johnson and Lee Benoy who have been most helpful and have tried to put me "back on the rails" at several points. All misunderstandings about the program that survive in this document are of course my own responsibility. Thanks also go to Alicia Chellis, Ted Dolotta, Debbie Haley, Tom London, Cathy Perez, John Reiser, and Bruce Rowland for their encouragement and assistance.

John Lions

Table of Contents

Chapter 1: Introduction 1
- 1.1 The Present Work 1
- 1.2 Source Code Files 2
- 1.3 Editorial Changes 2
- 1.4 Other Comments 3

Chapter 2: The Header Files 7
- 2.1 The File "manifest" 7
  - 2.1.1 Operators 7
  - 2.1.2 Operator Groups 7
  - 2.1.3 Condition Names 9
  - 2.1.4 Trees 9
  - 2.1.5 Operand Types 9
  - 2.1.6 `manifest` miscellany 11
- 2.2 The File "macdefs" 11
  - 2.2.1 `AUTOINIT`, `ARGINIT` 13
  - 2.2.2 `macdefs` miscellany 13
- 2.3 The File "mac2defs" 13
  - 2.3.1 Registers 13
  - 2.3.2 `mac2defs` miscellany 13
- 2.4 The File "mfile2" 13
  - 2.4.1 Groups of Operators 15
  - 2.4.2 Cookies 15
  - 2.4.3 Shapes 15
  - 2.4.4 More Types 15
  - 2.4.5 Needs 17
  - 2.4.6 Reclamation Cookies 17
  - 2.4.7 Nodes 17
  - 2.4.8 Pot Pourri 19
- 2.5 Code Templates 21
- 2.6 Addressing Modes 21

Chapter 3: The File "common" 25
- 3.1 Error Messages 25
- 3.2 Tree Nodes 25
- 3.3 Tree Walks 25
  - 3.3.1 `walkf` 25
  - 3.3.2 `fwalk` 27
- 3.4 The `dope` arrays 27
- 3.5 `mkdope` 27
- 3.6 `tprint` 27

Chapter 4: The File "reader.c" Part One 31
- 4.1 Variables 31
- 4.2 `p2init` 31
- 4.3 `main` 33
- 4.4 `rdin` 37
- 4.5 `eread` 37

4.6 `eprint` 39
4.7 `delay` 39
4.8 `delay1` 39
4.9 `delay2` 43

Chapter 5: The File "reader.c" Part Two 45
5.1 `codgen` 45
5.2 `canon` 45
5.3 `store` 47
5.4 `stoarg` 49
5.5 `markcall` 49
5.6 `constore` 51
5.7 `prcook` 51
5.8 `rcount` 51

Chapter 6: The File "reader.c" Part Three 52
6.1 Comparison with `codgen` 52
6.2 Strategy 52
6.3 Code Sections 53
6.4 First Section 53
6.5 Second Section 55
6.6 Third Section 55
6.7 Conditional Operators 57
6.8 Some Miscellaneous Cases 59
6.9 Procedure Calls 59
6.10 Fourth Section 62

Chapter 7: The File "reader.c" Part Four 63
7.1 `negrel` 63
7.2 `cbranch` 63
7.3 `ffld` 67
7.4 `oreg2` 69

Chapter 8: The File "match.c" 73
8.1 `setrew` 73
8.2 `match` 75
8.3 `getlr` 77
8.4 `tshape` 77
8.5 `ttype` 79
8.6 `expand` 81

Chapter 9: The File "allo.c" 85
9.1 Declarations 85
9.2 `allo0` 87
9.3 `allchk` 87
9.4 `allo` 87
9.5 Free Registers 89
9.5.1 `freereg` 89
9.5.2 `usable` 89
9.5.3 `shareit` 89
9.5.4 `ushare` 91
9.6 `freetemp` 91
9.7 `reclaim` 91
9.8 `rwprint` 93
9.9 `recl2` 93
9.10 `rfree` 93
9.11 `rbusy` 93
9.12 `ncopy` 95

9.13 tcopy 95
9.14 Keeping busy 95

Chapter 10: The File "order.c" Part One 97
10.1 deltest 99
10.2 stoasg 99
10.3 mkadrs 99
10.4 rallo 101
10.5 mkrall 103

Chapter 11: The File "order.c" Part Two 105
11.1 sucomp 107
11.2 zum 109

Chapter 12: The File "order.c" Part Three 111
12.1 getlab 111
12.2 deflab 111
12.3 offstar 111
12.4 The "set" procedures 111
12.4.1 setincr 111
12.4.2 setstr 111
12.4.3 setasop 111
12.4.4 setasg 113
12.4.5 setbin 113
12.5 niceuty 115
12.6 notoff 115
12.7 genargs 115
12.8 argsize 115

Chapter 13: The File "local2.c" Part One 117
13.1 Declarations 117
13.2 setregs 119
13.3 eobl2 119
13.4 lineid 119
13.5 where 119
13.6 hardops 123
13.7 optim2 123
13.8 myreader 125
13.9 cbgen 125
13.10 callreg 125
13.11 genscall 127
13.12 gencall 127
13.13 popargs 127

Chapter 14: The File "local2.c" Part Two 129
14.1 nextcook 129
14.2 lastchance 131
14.3 rewfld 131
14.4 spsz 131
14.5 szty 131
14.6 shltype 131
14.7 shumul 131
14.8 special 131
14.9 shtemp 131
14.10 flshape 131
14.11 acon 131
14.12 adrcon 133
14.13 adrput 133

14.14 conput 133
14.15 insput 133
14.16 upput 133
14.17 rmove 135
14.18 hopcode 135
14.19 zzzcode 135

Chapter 15: The File "table.c" 141
15.1 Macro Expansion 141
15.2 Table Searching 141
15.3 Some Statistics 143
15.3.1 Template Operators 143
15.3.2 Operator Summary 145
15.3.3 Visit Summary 145
15.3.4 Shape Summary 145
15.4 Some Comments 145

Chapter 16: Conclusion 157

Appendix A. Cross-reference 158

Appendix B. Defined Symbols 166

Appendix C. Procedure Calls Arranged by Caller 171

Appendix D. Procedure Calls Arranged by Callee 173

Chapter 1: Introduction

This document attempts a detailed examination of the source code for the second pass of the PDP11 version of the Portable C compiler.

The package of programs that the user regards as the "compiler" includes a pre-processor and a post-optimizer/assembler. Within the compiler proper, the first pass performs lexical and syntactical analysis of the source program, performs some storage allocation and generates specific code for procedure entry and exit points, and for `switch` statements. It builds binary trees to represent expressions that are to be evaluated. These trees are written to an intermediate file that is subsequently read back by the second pass of the compiler. The latter takes the trees and massages them in various ways, until code can be generated. (There is also a one-pass version of the compiler in which the expression trees are built and then broken down immediately. This version is somewhat larger, but it is also significantly faster than the two-pass version, because the overhead of writing and reading the intermediate file is eliminated.)

Thus, the principal task of the second pass is to take expression trees generated in the first pass, and to reduce these to assembler code. The goal of the implementation is to produce code that is locally optimum in the sense of minimizing the number of intermediate values that must be stored outside the processor's high-speed, readily accessible registers. The compiler applies heuristic rules based on theory given by Sethi and Ullman for finding the optimal assignment in a simplified situation (see, for example, *Principles of Compiler Design*, by A. V. Aho and J. D. Ullman, Addison-Wesley, 1977, p. 537). These rules determine when intermediate results must be stored outside the processor's registers in the object-time stack area. Attempts are made to break the original tree into a forest of trees, each of which can be processed independently. The compiler also attempts to take as much advantage as possible of the situations where address calculations can be carried out implicitly via hardware (i.e. the use of index and base registers).

1.1 The Present Work

The present document deals primarily with the second pass of the compiler because the time available to the present writer was not sufficient to cover the whole compiler and because:

1. The second pass is more machine-dependent than the first pass, and hence is of more interest to those people who are actively involved in transferring the Portable C compiler to other processors.

2. Whereas lexical analysis and syntactical analysis are now fairly well understood from a theoretical viewpoint, the code-generation phase of compilers is not so well understood, and thus constitutes one of the more interesting parts of the compiler.

3. The second pass of the Portable C compiler is also used as the second pass of the Fortran 77 compiler written by S. I. Feldman.

4. From a pedagogic point of view, the attraction of treating the compiler in terms of two separate passes was obvious, and the second pass seemed like a good place to start.

The PDP11 version was chosen because it is likely to be the most widely distributed version, at least in the near future, and also because it relates to a machine whose characteristics are widely understood.

1.2 Source Code Files

The approach that has been adopted here and that is based on previous favorable experience, is to present an edited version of the actual source code (the version is a snapshot taken in November, 1978). This is accompanied by amplifying and explanatory comments intended to guide the reader over the rougher spots, and to help him or her gain an understanding of the program, if not in just one pass through the source code, then in substantially fewer than might otherwise be needed.

The source code for the Portable C compiler exists as a set of files, of which the following are relevant to the second pass:

```
manifest   common     order.c
macdefs    reader.c   local2.c
mac2defs   match.c    table.c
mfile2     allo.c
```

The first four files (the first column) are header files that are "included" by the remaining files during compilation. (In practice, these remaining files "include" `mfile2`, which in turn "includes" `manifest`, `macdefs` and `mac2defs`.) Of these, `macdefs` and `mac2defs` are machine-dependent.

The next group of four files (the second column) are files that are considered to be machine-independent, i.e. the same in all versions of the compiler. The remaining three files contain the parts of the code that are expected to be different for each different machine type.

The job of transporting the compiler to a new machine consists largely of modifying, adapting and changing five files, together with the two machine-dependent files from the first pass, `code.c` and `local.c`.

1.3 Editorial Changes

Although the working version of the source code is really quite clean from a documentation viewpoint, the effort to prepare the version that appears in this document has been considerable, and should not be underestimated. The value of a careful presentation of the code may be reckoned differently by different individuals, but it is the the conviction of the present writer that it is highly important.

In editing the source code, lines that were too long were shortened, usually by breaking them into two. (This is not difficult, but it is time consuming. It is hard to see how a mechanical procedure could be used to do the job and still give results that are aesthetically acceptable in all cases.) Each source code line has been labeled with a unique four digit number and padding has been added to mark more prominently the end of each procedure. The four digit number provides a convenient means for cross-referencing within the text. Thus, for example, a reference to "`cbranch` (1832)" is intended to direct the reader to line 1832, which occurs in the procedure `cbranch`.

The contents of files have been re-arranged, in some cases quite extensively, to allow the presentation to flow more logically. In general, the policy has been to order the procedures in a "top-down" manner, i.e. so that the code for a procedure occurs after the first call on the procedure has appeared. The general plan for the text of this document has been to follow the source code through in the order in which it is presented. Thus, in general, there should be no difficulty in correlating code with comments.

The remaining editorial change of importance that needs to be mentioned is the omission of parts of various files that refer only to the one-pass version of the compiler. Since these are, in general, a re-statement of things that are already said, it was felt that they could be dispensed with here.

Also important to state are some of the things that were *not* changed. No variable names were changed, though the temptation to do so at times was very strong. The naming of variables, for better or worse, should remain the program author's responsibility. Likewise, it was felt that the movement of procedures between files would be too radical a change, since it would cause difficulty for readers who wish eventually to work with the code in practical situations. Thus procedures such as `ncopy` and `tcopy` still appear in `order.c`, although they would be very much more at home with the other tree manipulation routines in the file `common`. (There is a reason for this, of course: these procedures are needed only in the second pass, whereas the procedures in `common` are used in both passes.)

If some particular pleas to prospective program writers can be made appropriately at this point, they would be to:

1. Take care with the physical layout of your program. (As well as observing sensible indentation rules, do not allow the right hand margins of your code to wander much beyond column 65.)
2. Think long and hard about the choice of variable names. (For example, the practice of naming the subfields of a given structure with the same initial prefix can be more useful to the reader than choosing names that are always euphonious.)
3. Take care to arrange procedures and variable declarations among a set of files in a way that is consistent with some logical criterion.

The usual admonitions about lacing the code with an ample, but not too generous supply of relevant and well-positioned comments still apply, of course.

Another matter of concern for documentation has been the provision of various machine-generated tables to supplement and support the source code. With this particular program, it seems that a completely general cross-reference would not be so useful as some more specialized tables, especially an alphabetical list of defined symbols, and tables showing caller-callee relationships for procedures, arranged both by caller and by callee.

1.4 Other Comments

The coding style within the Portable C compiler is generally consistent and clear. As with many programs, the principal difficulty for the reader is to understand the problem rather than its solution. It is fair to say that the age-old problem of providing the reader with an adequate supply of incisive, well-placed comments, is not solved here either.

The problem of dividing the source code into machine-dependent and machine-independent parts has been solved, in a sense, by dividing the material into files that are clearly labeled as machine-dependent and machine-independent. But many lines of the code in the machine-dependent parts are in fact common to all versions of the compiler, whereas substantial parts of the machine-independent parts exist to serve only one or a few machine types. The method most commonly practiced for exorcising the machine-dependent parts of the code from the machine-independent framework, namely the invention of special procedures, many of which are called only once, very often seems awkward and contrived. This is not intended so much as a criticism of the Portable C compiler as a comment on the limitations of the program-building tools that now exist. These problems with the Portable C compiler suggest further development of the C preprocessor.

In the defense of the authors of the Portable C compiler, it should be pointed out that some of the less happy features of the code are the result of *force majeure* rather than an expression of individuality. For example, the slate of `extern` declarations in `mfile2` is a result of the limitations of some assemblers (notably the one for the Honeywell 6000). Enumeration data types and fields are not used in the source code of the compiler due to the compiler writer's universal need to be conservative in actually using new language features.

The Portable C compiler is known to work well as far as compiling correct code is concerned. For the PDP11, the code produced is neither uniformly better, nor uniformly worse than the code produced by the C compiler written by Dennis Ritchie, though the speed of compilation is definitely inferior. Object modules tend to be about the same size or slightly larger.

```
# include <stdio.h>

        /* manifest constant file for the lex/yacc interface */
# define ERROR     1
# define NAME      2
# define STRING    3
# define ICON      4
# define FCON      5
# define PLUS      6
# define MINUS     8
# define MUL       11
# define AND       14
# define OR        17
# define ER        19
# define QUEST     21
# define COLON     22
# define ANDAND    23
# define OROR      24

        /* special interfaces for yacc alone */
        /* These serve as abbreviations of 2 or more ops:
        ASOP   =, = ops
        RELOP LE,LT,GE,GT
        EQUOP EQ,NE
        DIVOP DIV,MOD
        SHIFTOP     LS,RS
        ICOP  INCR,DECR
        UNOP  NOT,COMPL
        STROP DOT,STREF
        */
# define ASOP      25
# define RELOP     26
# define EQUOP     27
# define DIVOP     28
# define SHIFTOP   29
# define INCOP     30
# define UNOP      31
# define STROP     32

        /* reserved words, etc */
# define TYPE      33
# define CLASS     34
# define STRUCT    35
# define RETURN    36
# define GOTO      37
# define IF        38
# define ELSE      39
# define SWITCH    40
# define BREAK     41
# define CONTINUE  42
# define WHILE     43
# define DO        44
# define FOR       45
# define DEFAULT   46
# define CASE      47
# define SIZEOF    48
# define ENUM      49

        /* little symbols, etc., namely

        LP  RP  LC  RC  LB  RB  CM  SM
        (   )   {   }   [   ]   ,   :

        */

# define LP        50
# define RP        51
# define LC        52
# define RC        53
# define LB        54
```

The speed of the Portable C compiler has always been an issue, and several changes have been introduced during development to improve this aspect. The original lexical scanner (which is part of the first pass) was replaced. A version of the compiler that merges the two passes into one, thus eliminating, the encoding, writing, reading and decoding of the intermediate file, is 30% larger but also substantially faster. Detailed examination of the code of the second pass has suggested many additional areas where speed improvements might be achieved. Further investigation is needed to determine which, if any, of the suggested improvements are likely to be worthwhile, but it does seem that with fine tuning, there is scope for substantially improving the execution speed of the compiler and thereby removing one of its perceived drawbacks.

```
# define RB         55
# define CM         56
# define SM         57
# define ASSIGN     58

      /* END OF YACC */

      /* left over tree building operators */
# define COMOP      59
# define DIV        60
# define MOD        62
# define LS         64
# define RS         66
# define DOT        68
# define STREF      69
# define CALL       70
# define FORTCALL   73
# define NOT        76
# define COMPL      77
# define INCR       78
# define DECR       79
# define EQ         80
# define NE         81
# define LE         82
# define LT         83
# define GE         84
# define GT         85
# define ULE        86
# define ULT        87
# define UGE        88
# define UGT        89
# define SETBIT     90
# define TESTBIT    91
# define RESETBIT   92
# define ARS        93
# define REG        94
# define OREG       95
# define CCODES     96
# define FREE       97
# define STASG      98
# define STARG      99
# define STCALL     100

      /* some conversion operators */
# define FLD        103
# define SCONV      104
# define PCONV      105
# define PMCONV     106
# define PVCONV     107

      /* special node operators, used for special contexts */
# define FORCE      108
# define CBRANCH    109
# define INIT       110
# define CAST       111

      /* operator modifiers */
# define ASG        1+
# define UNARY      2+

# define NOASG      (-1)+
# define NOUNARY    (-2)+
/* ----------------------------------------------------- */

      /* node types */
# define LTYPE      02
# define UTYPE      04
# define BITYPE     010

```

```
        /* operator information */
# define TYFLG    016
# define ASGFLG   01
# define LOGFLG   020
# define SIMPFLG  040
# define COMMFLG  0100
# define DIVFLG   0200
# define FLOFLG   0400
# define LTYFLG   01000
# define CALLFLG  02000
# define MULFLG   04000
# define SHFFLG   010000
# define ASGOPFLG 020000
# define SPFLG    040000

        /* operator condition names */
# define optype(o) (dope[o]&TYFLG)
# define asgop(o)  (dope[o]&ASGFLG)
# define logop(o)  (dope[o]&LOGFLG)
# define callop(o) (dope[o]&CALLFLG)
/* ================================================== */

        /* type names, used in symbol table building */
# define UNDEF    0
# define FARG     1
# define CHAR     2
# define SHORT    3
# define INT      4
# define LONG     5
# define FLOAT    6
# define DOUBLE   7
# define STRTY    8
# define UNIONTY  9
# define ENUMTY   10
# define MOETY    11
# define UCHAR    12
# define USHORT   13
# define UNSIGNED 14
# define ULONG    15

        /* type modifiers */
# define PTR      020
# define FTN      040
# define ARY      060

        /* type packing constants */
# define TMASK    060
# define TMASK1   0300
# define TMASK2   0360
# define BTMASK   017
# define BTSHIFT  4
# define TSHIFT   2

        /* macros   */
                                  /* set basic type of x to y */
# define MODTYPE(x,y)  x = (x&(~BTMASK))|y
# define BTYPE(x)      (x&BTMASK)   /* basic type of x */
# define ISUNSIGNED(x) ((x)<=ULONG&&(x)>=UCHAR)
# define UNSIGNABLE(x) ((x)<=LONG&&(x)>=CHAR)
# define ENUNSIGN(x)   ((x)+(UNSIGNED-INT))
# define DEUNSIGN(x)   ((x)+(INT-UNSIGNED))
# define ISPTR(x)      ((x&TMASK)==PTR)
# define ISFTN(x)      ((x&TMASK)==FTN) /* is x a function type */
# define ISARY(x)      ((x&TMASK)==ARY) /* is x an array type */
# define INCREF(x)     (((x&~BTMASK)<<TSHIFT)|PTR|(x&BTMASK))
# define DECREF(x)     (((x>>TSHIFT)&~BTMASK)|(x&BTMASK))
/* ================================================== */
```

The first four files of the program contain definitions for many of the symbols and structures, together with forward declarations for most of the variables, used by the program. The files are:

1. `manifest`: Machine-independent definitions, many of which are used in both passes of the compiler.
2. `macdefs`: Machine-dependent definitions needed in the first pass of the compiler. Some of these are also needed by the second pass.
3. `mac2defs`: Machine-dependent definitions needed in the second pass.
4. `mfile2`: Definitions for symbols used in the code templates, declarations for the node and template structures, and various forward declarations.

The fourth file, `mfile2`, "includes" each of the first three. In turn, `mfile2` is "included" by each of the other compilable files. Accordingly the scope of the definitions in these four files is the whole program. (In passing it may be noted that `stdio.h` is included by `manifest` at line 0001.)

2.1 The File "manifest"

2.1.1 Operators. `manifest` begins (lines 0001 to 0125) with definitions for a sequence of approximately one hundred *operator* types, about half of which are of interest only in the first pass of the compiler.

The particular association of numeric values with operator types is largely arbitrary. Since the compiler contains many `switch` statements that are keyed on an operator variable, there may be prospects for gains in code compactness and/or execution speed by fine-tuning these assignments. (Such prospects would be less if more elaborate techniques were employed by the compiler in the generation of code for `switch` statements, as is done by the regular C compiler.) Every node of an expression tree has an associated operator type, which is one of the values given on lines 0004 through 0125. Note that the value FREE (0109) is used to label nodes that are not currently assigned.

There are certain derived operators which are not given explicitly in the above mentioned list, but are created via the "macro operators" defined on lines 0128 to 0132. The most commonly occurring example of such an operator is UNARY MUL, whose value is 2 + 11 = 13. It will be seen readily that no other operator type has been assigned that value. (In passing it may be noted that NOASG and NOUNARY are not used in the second pass.)

2.1.2 Operator Groups. Operators sharing a common characteristic can be grouped in various ways. Unfortunately the bits and pieces used to define such groups in this program are scattered over three different files, `manifest`, `mfile2` and `common`. As we shall see later, the procedure `mkdope` (0811) constructs an array `dope` (0724) that contains a bitmask for each operator. This bit mask consists of:

1. an "assignment" flag (one bit).
2. a "type" field (three bits).
3. various other flags which are given on lines 0140 to 0153.

```
      /* table sizes    */
# define DSIZE   CAST+1 /* size of the dope array */
# define BCSZ       100 /* size of table to save break
                              and continue labels */
# define SYMTSZ     450 /* size of the symbol table */
# define DIMTABSZ   750 /* size of the dimension/size table */
# define PARAMSZ    100 /* size of the parameter stack */
# define SWITSZ     250 /* size of switch table */

# ifndef FORT           >>>>>>>>>>>>>>>>>>>>>>>>
# define TREESZ     350 /* space for building parse tree */
# else                  xxxxxxxxxxxxxxxxxxxxxxxx
# define TREESZ    1000
# endif                 <<<<<<<<<<<<<<<<<<<<<<<<
/* ================================================== */

      /* advance x to a multiple of y */
# define SETOFF(x,y)    if( x%y != 0 ) x = ( (x/y + 1) * y)
      /* can y bits be added to x without overflowing z */
# define NOFIT(x,y,z)   ( (x%z + y) > z )

      /* pack & unpack field descriptors (size & offset) */
# define PKFIELD(s,o)   ((o<<6)|s)
# define UPKFSZ(v)      (v&077)
# define UPKFOFF(v)     (v>>6)

      /* miscellaneous */
# define NOLAB    (-1)
# define TNULL     PTR   /* pointer to UNDEF */
# define NCHNAM      8  /* number of characters in a name */
/* -------------------------------------------------- */

typedef union ndu NODE;
typedef unsigned int TWORD;

      /* common defined variables */
extern int nerrors; /* number of errors seen so far */
extern NODE *NIL; /* a pointer which will always have 0 in it */
extern int dope[]; /* a vector containing operator information */
extern char *opst[]; /* a vector containing names for ops */
```

These complicating type modifications may be cascaded. Each level requires another two bit field in the operand type word. There is a set of macros, given on lines 0195 to 0205, for encoding and extracting this information. The important ones to notice for the second pass are:

1. `BTYPE`: extract the basic type.
2. `ISPTR`: is this a pointer type?
3. `ISFTN`: is this a function type?
4. `ISARY`: is this an array type?

2.1.6 `manifest` *miscellany.* Most of the remaining material in the file `manifest` is adequately commented. The following are worthy of notice at this juncture:

0225: `SETOFF` is an expression whose value is that of its first argument rounded up to a multiple of its second argument. It is used primarily in the calculation of byte and word offsets from bit offset values.

0230: `PKFIELD` is used in the first pass to store information in the `rval` field of a `NODE` structure regarding the size and offset of bit-fields in structures. This information is subsequently retrieved using `UPKFSZ` and `UPKFOFF` respectively.

0237: `NCHNAM` is the size of the character array in each `NODE` structure, i.e. it defines the maximum length of unique variable names.

0240: `NODE` (0240) is the type for the building blocks or nodes for the expression trees.

0241: `TWORD` (0241) is the variable type that stores operand type information. One such variable is part of every `NODE`.

2.2 The File "macdefs"

This file, which begins at line 0248, gives machine-dependent parameter definitions that are needed in the first pass of the compiler. Some of these are also needed in the second pass.

0250: Space is allocated in units of one bit. The different operand types each have associated, hardware-determined sizes which reduce to one or a combination of the sizes of:

1. character.
2. short or long integer.
3. single or double floating point.
4. address constant (or pointer).

It may be noted that whereas offset calculations in the first pass of the compiler are conducted entirely in bits, the calculations in the second pass are largely in terms of addressable storage units.

0258: Likewise, the hardware for most machines dictates very strongly what the alignment boundaries must be*. (It is assumed that the reader is familiar with the implications of aligning characters to an eight bit boundary, and short integers, to a sixteen bit boundary, etc.)

* On the VAX11/780, the architecture allows alignment for all operand types, to eight bit boundaries. However the hardware implementation exacts a significant run-time penalty if the operand types are not aligned to their "natural" boundaries. Thus for this machine, there is a potential space/speed tradeoff that different VAX11/780 installations may prefer to solve differently.

```
/*    PDP11 Values */

# define SZCHAR      8
# define SZINT       16
# define SZFLOAT     32
# define SZDOUBLE    64
# define SZLONG      32
# define SZSHORT     16
# define SZPOINT     16

# define ALCHAR      8
# define ALINT       16
# define ALFLOAT     16
# define ALDOUBLE    16
# define ALLONG      16
# define ALSHORT     16
# define ALPOINT     16
# define ALSTRUCT    16
# define ALSTACK     16

# define ARGINIT     32
# define AUTOINIT    48

        /*  size in which constants are converted */
        /*  should be long if feasable */
# define CONSZ       long
# define CONFMT      "%Ld"

        /*  size in which offsets are kept
        /*  should be large enough to cover address space in bits */
# define OFFSZ       long

        /*  character set macro */
# define CCTRANS(x) x

        /*  register cookie for stack poINTer */
# define STKREG      5
# define ARGREG      5

        /*  maximum and minimum register variables */
# define MAXRVAR     4
# define MINRVAR     2

        /* various standard pieces of code are used */
# define STDPRTREE
# define LABFMT      "L%d"

        /*  definition indicates automatics and/or temporaries
                    are on a negative growing stack */
# define BACKAUTO
# define BACKTEMP
# define RTOLBYTES
# define ENUMSIZE(high,low) INT

# define makecc(val,i)   lastcon = i ? (val<<8)|lastcon : val
```

2.2.1 AUTOINIT, ARGINIT. The Portable C compiler provides a general mechanism for building the run-time stack frames needed by procedures. The issues involved are discussed in the internal technical memorandum, *"The C Language Calling Sequence"*, by S.C. Johnson, D.M. Ritchie and M.E. Lesk.

AUTOINIT defines the growth (in bits) of the stack, beyond the point indicated by the frame pointer, due to the storage of CPU registers at procedure entry time. (The frame pointer marks the beginning, or some point offset by a standard amount from the beginning of the stack frame.) On the PDP11, where the frame pointer is R5, the stack growth is three words (48 bits) to store values of R4, R3 and R2.

ARGINIT is not used in the second pass, at least for the PDP11. It is intended for use with a separate "argument pointer", which is needed, for example, when the arguments passed to a procedure are not stored in a location fixed relative to the frame pointer.

2.2.2 macdefs *miscellany.* Many of the declarations in macdefs are not relevant to the second pass. Of those given from line 0271 on, MAXRVAR and MINRVAR are relevant to the allocation of temporary registers (they define the range of registers which may be preempted for local variables in fact), BACKTEMP (0298) specifies that temporary storage is allocated backwards in memory, and RTOLBYTES (0299) is used to flag the relatively unusual byte ordering of the PDP11.

2.3 The File "mac2defs"

This file, which begins at line 0303, contains machine-independent definitions, additional to those given in macdefs, which are needed in the second pass of the Portable C compiler.

2.3.1 Registers. There are assumed to be two different classes of registers which can be used in the evaluation of expressions, and which the compiler must assign.

In the PDP11 version, type A registers are general registers which can store integers and pointers, and which are generally in demand and in short supply. On the other hand, type B registers are floating point registers for which the supply is reasonably adequate, and allocation is no great problem. In retrospect, it seems* that it would have been preferable to treat the floating point registers as additional type A registers, rather than as a different species, as is done in the regular compiler for the PDP11.

The concept of "B" registers was introduced into the compiler, to accommodate the index registers of the Honeywell computer. One of the deficiencies** of the present compiler is its inability to recognize and handle more than two distinct types of registers.

2.3.2 mac2defs *miscellany.* SAVEREGION and wdal are not used in the second pass of the PDP11 compiler. The defined symbol MYREADER is used, in effect, to indicate that a procedure, myreader (3926), exists, and is to be invoked (in main at line 1031). The variable fltused (0349) is used for the PDP11 to set a flag, which will effect the loading of library routines with the compiled program.

The defined symbols, STOFARG, STOARG and STOSTARG, all stand for procedures which may be optionally present, and which are called by store (1325) to take machine-dependent actions appropriately for the generation of code to calculate argument values. No special actions are required on the PDP11, so these symbols have null values.

2.4 The File "mfile2"

This file contains various machine-independent definitions and declarations of global significance to the second pass of the Portable C compiler. There is a companion file, mfile1, which plays a similar role in the first pass of the compiler.

* Communication from Lee Benoy.

** Communication from Steve Johnson.

```
/*    PDP11 Registers */

      /* scratch registers */
# define R0 0
# define R1 1

      /* register variables */
# define R2 2
# define R3 3
# define R4 4

      /* special purpose */
# define R5 5     /* frame pointer */
# define SP 6     /* stack pointer */
# define PC 7     /* program counter */

      /* floating registers */
# define FR0 8
# define FR1 9
# define FR2 10
# define FR3 11
# define FR4 12
# define FR5 13

# define SAVEREGION 8 /* number of bytes for save area */

# define BYTEOFF(x) ((x)&01)
# define wdal(k) (BYTEOFF(k)==0)
# define BITOOR(x) ((x)>>3)  /* bit offset to oreg offset */

# define REGSZ 14

# define TMPREG R5

# define STOARG(p)  /* just evaluate the arguments,
                                     and be done with it... */
# define STOFARG(p)
# define STOSTARG(p)
# define genfcall(a,b) gencall(a,b)

      /* shape for constants between -128 and 127 */
# define SCCON (SPECIAL+100)
      /* shape for constants between 0 and 32767 */
# define SICON (SPECIAL+101)

# define MYREADER(p) myreader(p)
extern int fltused;

      /* calls can be nested on the PDP-11 */
# define NESTCALLS
```

2.4.1 Groups of Operators.

0360: In coding the set of operator templates in the array `table`, it is convenient and possible to provide some templates which apply for a whole group of operators. Some such groups are implied by the names given on lines 0360 to 0370. The ASG (0128) operator can also be applied to these to produce e.g. ASG OPLOG, which has a value of 010017.

0375: MNOPE (0375), MDONE (0376) are values returned by the procedure `match` (2159) when it has been decided that the situation is either hopeless, or completely under control, respectively.

2.4.2 Cookies. In the present context, the term "cookie" (see line 0379) means "goal" or "set of alternative goals". Each expression tree represents a calculation that may be carried out to yield a result. The cookie refers to the disposition of this result. In particular, the cookie FOREFF implies that all results of the calculation that are left in the processor registers and in the temporary part of the object-time stack may be discarded. All useful results of the calculation will have already been saved explicitly. All trees passed from the first to the second pass of the compiler are to be computed "for effect" only.

In the case of subtrees, even if the overall goal is "for effect", the result of the subtree calculation may be temporarily important and must be saved somewhere. Just where may depend on what other registers are already being used. Failing all else, the result may be placed in the temporary part of the object time stack. (This is generally undesirable because access to stack locations is slower, and the necessary code is longer, than for reference to the processor registers.)

The other goals are listed, with comments, on lines 0382 to 0389. Note that references to `lvalue` on lines 0384, 0385 and elsewhere certainly do not apply in the case of the PDP11.

2.4.3 Shapes. The use of the term "shape" in the present context is somewhat unconventional. It is used to suggest the way an operand, represented by a particular subtree, can be accessed. It may be in an A register (SAREG, 0395), or in a temporary A register (STAREG, 0396), or in a B register (STBREG, 0398), or in the condition codes (SCC, 0399). It may be a constant (SCON, 0401), or a subfield of a word (SFLD, 0402).

The operand may be accessed indirectly through a pointer variable (STARNM, 0404), or through a register pointer (STARREG, 0405).

The reader should be careful to distinguish between STAREG and STARREG. They are to be "parsed" quite differently, as "S-T-AREG" and "STAR-REG", respectively; furthermore, they must be distinguished from the operator type STARG (0111)*

There are also a number of special "shapes" for constants: SZERO (0408), SONE (0409) and SMONE (0410). The latter, "minus one", is not handled specially on the PDP11. On the other hand, short integer constants may be given special treatment in some circumstances on the PDP11 (see lines 0344 and 0346). SWADD (0406), meaning "shape of word address", is relevant to the Honeywell 6000.

2.4.4 More Types. The definitions which begin on line 0417 are for a set of operand types. Unlike the set previously given on lines 0163 to 0178, which were designed to be compactly encoded in a four bit field, these definitions are for a set of bit masks which may be combined into sets of alternatives.

* These are to be read as "shape of a temporary A register", "indirection through a register" and "structure argument", respectively. Note also that the meaning of STARREG in the current PDP11 version is slightly non-standard, since it can refer to autoincrement and autodecrement addressing modes.

```
# include "macdefs"
# include "mac2defs"
# include "manifest"

        /* OP descriptors */
        /* the ASG operator may be used on some of these */

# define OPSIMP     010000  /* +, -, &, |, ^ */
# define OPCOMM     010002  /* +, &, |, ^ */
# define OPMUL      010004  /* *, / */
# define OPDIV      010006  /* /, % */
# define OPUNARY    010010  /* unary ops */
# define OPLEAF     010012  /* leaves */
# define OPANY      010014  /* any op... */
# define OPLOG      010016  /* logical ops */
# define OPFLOAT    010020  /* +, -, *, or / (for floats) */
# define OPSHFT     010022  /* <<, >> */
# define OPLTYPE    010024  /* leaf type nodes (e.g, NAME, ICON) */
/* -------------------------------------------------- */

        /* match returns */

# define MNOPE      010000
# define MDONE      010001
/* -------------------------------------------------- */

        /* cookies, used as arguments to codgen */

# define FOREFF     01    /* compute for effects only */
# define INAREG     02    /* compute into a register */
# define INTAREG    04    /* compute into a scratch register */
# define INBREG     010   /* compute into a lvalue register */
# define INTBREG    020   /* compute into a scratch lvalue register */
# define FORCC      040   /* compute for condition codes only */
# define INTEMP     010000  /* compute into a temporary location */
# define FORARG     020000  /* compute for an argument of a function */
# define FORREW     040000  /* search the table for a rewrite rule */
/* -------------------------------------------------- */

        /* shapes */

# define SANY       01    /* same as FOREFF */
# define SAREG      02    /* same as INAREG */
# define STAREG     04    /* same as INTAREG */
# define SBREG      010   /* same as INBREG */
# define STBREG     020   /* same as INTBREG */
# define SCC        040   /* same as FORCC */
# define SNAME      0100
# define SCON       0200
# define SFLD       0400
# define SOREG      01000
# define STARNM     02000
# define STARREG    04000
# define SWADD      040000
# define SPECIAL    0100000
# define SZERO      SPECIAL
# define SONE       (SPECIAL|1)
# define SMONE      (SPECIAL|2)

/* FORARG, INTEMP are carefully not conflicting with shapes */
/* -------------------------------------------------- */
```

2.4.5 Needs. Most of the definitions in the section of code beginning at line 0381 are for items which can occur in the code templates. The formal declaration of the structure which encodes a single template occurs at line 0539, and is discussed in more detail below.

The particular set of definitions that commence at line 0435 under the label "Needs", refer to the resources which may be needed temporarily during the sequence of instructions defined by the template. For example, `NAREG` (0435) specifies that a temporary A register will be required, and one such register must be made available if the code sequence specified by the template is to be used.

`NASL` (0438) specifies that a temporary A register is needed, but that this can be the same register as used by the "left hand" operand *provided* the contents of this register do not have to be kept intact for some other reason.

`NACOUNT` (0436) is a mask to define the field in which the number of A registers is encoded. `NAMASK` (0437) is used to isolate the requests (by masking out other fields) for A registers from other requests (e.g. for B registers). These are all used by the procedure `allo` (2493).

`REWRITE` is a special need, which should be encountered when there is no hope of matching the particular node with any of the regular templates. It signals that the tree will have to be remodeled before the template matching should be attempted again.

0449: `MUSTDO` and `NOPREF` are used to qualify the value of the `rall` field in the `ndu` structure ... see line 0469.

2.4.6 Reclamation Cookies. A set of definitions for these begins on line 0455. After a template has been matched, and the appropriate instructions emitted, the tree must be rewritten to replace the matched subtree by, typically, a single node representing the result obtained. The "reclamation cookie" is used to denote where the result may be found. In many cases, the result is available in more than one location, e.g., after a move instruction, so that the practical problem becomes to decide which of these will be most convenient.

The cookies are bitmasks that may be combined to represent multiple alternatives. `RLEFT` (0456) denotes that the result will be in the left operand of a binary pair. `RESC1`, etc. denotes the first, etc. temporary registers assigned. `RNULL` denotes that no result need be saved, whereas `RNOP` denotes that there is *no* result to be saved.

2.4.7 Nodes. The type definition on line 0240 equates, for the second pass, the type NODE (which is frequently used) with the type `ndu` (which is not used otherwise). The `NODE` type is specified differently during the first pass of the two pass version of the compiler, and differently again in the single pass version.

The type `ndu` is a union of four different structures, which are declared beginning at line 0465. All four structures have their first four fields in common:

1. `op` is an operator type.
2. `rall` is used for expressing preferences for where (in which register) results should be stored.
3. `type` describes the associated operand type.
4. `su` expresses the number of registers needed during the calculation represented by the subtree.

Further the first two forms, A and B, which may be associated with `BITYPE` and `LTYPE` nodes respectively, have a common fifth field, `name`, whose contents, when non-null, are derived from a variable name in the source program. The structure for `UTYPE` nodes does not appear explicitly, but is in effect an amalgam of forms A and B, with a right "value" and a left "node pointer".

```
        /* types */

# define TCHAR      01
# define TSHORT     02
# define TINT       04
# define TLONG      010
# define TFLOAT     020
# define TDOUBLE    040
# define TPOINT     0100
# define TUCHAR     0200
# define TUSHORT    0400
# define TUNSIGNED  01000
# define TULONG     02000
# define TPTRTO     04000   /* pointer to one of the above */
# define TANY       010000  /* matches anything within reason */
# define TSTRUCT    020000  /* structure or union */
/* -------------------------------------------------- */

        /* needs */

# define NAREG      01
# define NACOUNT    03
# define NAMASK     017
# define NASL       04   /* share left register */
# define NASR       010  /* share right register */
# define NBREG      020
# define NBCOUNT    060
# define NBMASK     0360
# define NBSL       0100
# define NBSR       0200
# define NTEMP      0400
# define NTMASK     07400
# define REWRITE    010000

# define MUSTDO     010000  /* force register requirements */
# define NOPREF     020000  /* no preference for register assignment */
/* -------------------------------------------------- */

        /* reclamation cookies */

# define RNULL      0       /* clobber result */
# define RLEFT      01
# define RRIGHT     02
# define RESC1      04
# define RESC2      010
# define RESC3      020
# define RESCC      04000
# define RNOP       010000  /* DANGER: can cause loops.. */
/* ================================================== */

```

0498: The sizes of structures, and their alignments, are given in multiples of characters.

2.4.8 Pot Pourri. The latter part of `mfile2` (lines 0505 to 0593) is a bit of a mixture (to put it mildly). It consists mainly of forward declarations for variables which are declared elsewhere. In view of the way `mfile2` is used, it would seem better to replace most of these forward declarations by the actual declarations. However, as has already been noted, the present arrangement has been dictated by the limited capacity of some assemblers to handle globally defined variables. Descriptions for many of these variables will be given again when they are re-encountered. However several are worthy of comment now.

The first group of variables (from line 0505 to 0517) are concerned with `NODE`s.

1. The array `node` (0510) is the basic supply of structures from which trees are built.
2. The array `resc` (0511) is used to hold information, at the time code is being generated, about the temporary storage and registers.
3. `deltrees` (0508) is an array of node pointers, used to keep track of subtrees that have been broken off from the main tree by `delay2` (1233), and which await later processing. (The size of this array seems to be very generous.)
4. The integer `deli` (0506) keeps track of the latest entry in `deltrees`.
5. The procedures `talloc`, `eread`, `tcopy` and `getlr` all return a node pointer as their result.

0521: `rstatus` (3717) is a constant, machine-dependent data array, which is declared and initialized in the file `local2.c`. It gives information about the type and status of individual processor registers.

0522: `busy` (2453) is used to keep track of the usage of temporary registers during expression evaluation.

0524: `respref` is both the name of a structure (defined here) and the name of an array of such structures. The latter is initialized beginning at line 3729. It is used in selecting the best of a set of alternative results from the execution of a particular machine instruction.

0532: `SETSTO` is a macro which assigns values to `stocook` and `stotree`. These values are determined by `store` (1325) as it attempts to decide which subtree should be worked upon next.

0539: See the next section below.

0553: Offsets and related quantities are reckoned in bits, so they are stored and manipulated as long integers. (See the definition of `OFFSZ` (0278).)

0561: `nrecur` is reinitialized to zero for each expression tree, and is incremented at each call of `order` (1537) and `match` (2168).

0563: If the value of `nrecur` reaches `NRECUR`, the compiler exits on the assumption that it is looping infinitely (see line 1517).

0589: These remaining macros are used for machines such as the IBM 360/370 and the Interdata 8/32, which have "base-index" addressing. They are not needed for the PDP11.

```
union ndu {

	struct {	/* form A */
		int op;
		int rall;
		TWORD type;
		int su;
		char name[NCHNAM];
		NODE *left;
		NODE *right;
		};

	struct {	/* form B */
		int op;
		int rall;
		TWORD type;
		int su;
		char name[NCHNAM];
		CONSZ lval;
		int rval;
		};

	struct {	/* form C */
		int op, rall;
		TWORD type;
		int su;
		int label;  /* for use with branching */
		};

	struct {	/* form D */
		int op, rall;
		TWORD type;
		int su;
		int stsize; /* sizes of structure objects */
		int stalign;	/* alignment of structure objects */
		};

	};
/* -------------------------------------------------- */

# define DELAYS		20
extern int deli;  /* mmmmm */

extern NODE *deltrees[DELAYS];
extern NODE *stotree;
NODE node[TREESZ];
extern NODE resc[];

extern NODE
	*talloc(),
	*eread(),
	*tcopy(),
	*getlr();

	/* register allocation */

extern int rstatus[];
extern int busy[];

extern struct respref { int cform; int mform; } respref[];

# define isbreg(r)	(rstatus[r]&SBREG)
# define istreg(r)	(rstatus[r]&(STBREG|STAREG))
# define istnode(p)	(p->op==REG && istreg(p->rval))
```

2.5 Code Templates

Each code template contains a description of a subtree, or class of subtrees, plus a "recipe" for producing the assembler code that will carry out the calculation represented by that subtree.

The declaration of the data structure, `optab`, which will hold the description of a single template, and a forward declaration for the array `table` are given starting at line 0539. The components of `optab` are:

1. `op`, the type of node that may be matched.
2. `visit`, the type(s) of goal that can be met.
3. `lshape`, allowable shape(s) for the left subtree.
4. `ltype`, allowable operand type(s) for the left subtree.
5. `rshape`, `rtype`, ditto for the right subtree.
6. `needs`, resources that will be required (especially temporary registers).
7. `rewrite`, rule for rewriting the tree after a match has been made.
8. `cstring`, a character string which expands into a set of assembler instructions.

The file `table.c`, which begins at line 4664, declares and initializes the array `table`. Any serious student of the compiler will need to analyze this array at some length. For the moment, it will be worth looking at just a few parts of that array.

Beginning at line 4701, there is a template for a subtree whose root node has the operator `ASSIGN`. It is used to copy a long integer, signed or unsigned, from one directly addressable location to another. The shape and type restrictions for both the left and right subtrees are the same. `LWD` is defined on line 4667, and specifies a set of alternative shapes that are acceptable. No additional resources are needed, and after the operation, the "result" is accessible through either the left or right operands. The template may be used `FOREFF`, in which case the result is not of interest, or else it may be used for `INAREG`, i.e. to get the result into a pair of A registers. If neither the right or left subtrees represent a register pair, then additional move instructions may need to be generated.

The next template, which is initialized starting at line 4707, is very similar except for the shape of the left subtree, i.e. the destination. If the left subtree has the shape `STARNM`, i.e. contains a pointer to the destination, then this pointer can be brought into a temporary A register, and used to address the destination. The temporary register may be an additional register ("need" `NAREG`) or may reuse the register already used in the left subtree, provided no other use for that register already exists, i.e. if the left register is sharable ("need" `NASL`). In this case, the result (if required) will be more readily accessible as the right subtree ("reclamation cookie" is `RRIGHT`), since the left register will have been incremented.

2.6 Addressing Modes

One of the distinguishing features of the PDP11 class of computers and its successor, the VAX11/780, is a rich, flexible and somewhat complex set of addressing modes. An addressing mode is a convention for using the contents of a designated register, possibly combined with a word obtained from the instruction stream, to define the address of an operand.

The PDP11 has eight basic addressing modes and eight general purpose registers, so that a (mode, register) pair can be defined in a string of six bits. Two such fields can fit into a single sixteen bit word, so the PDP11 is able to encode a number of two address instructions efficiently. (In some of these, both operands are addressed via the general addressing modes, and in others, one of the operands is addressed in the general way, and the other must be a register.) Descriptions of the addressing modes for the PDP11 are given in the "PDP11 Processor Handbook" (Digital Equipment Corporation, various editions) and are not repeated here.

```
# define REGLOOP(i)      for(i=0;i<REGSZ;++i)

# define SETSTO(x,y)     (stotree=(x),stocook=(y))
extern int stocook;

extern int callflag;

extern int fregs;

extern struct optab {
        int op;
        int visit;
        int lshape;
        int ltype;
        int rshape;
        int rtype;
        int needs;
        int rewrite;
        char * cstring;
        }
        table[];


extern OFFSZ tmpoff;
extern OFFSZ maxoff;
extern OFFSZ baseoff;
extern OFFSZ maxtemp;

extern int maxtreg;
extern int ftnno;

extern int nrecur;      /* flag to keep track of recursions */

# define NRECUR    (10*TREESZ)

extern CONSZ rdin();

extern int eprint();

extern char *rnames[];

extern int lineno;
extern char filename[];
extern int fldshf, fldsz;
extern int lflag, xdebug, udebug, edebug, odebug,
                rdebug, radebug, tdebug, sdebug;

# ifndef EXIT           >>>>>>>>>>>>>>>>>>>>>>>>>>>
# define EXIT           exit
# endif                 <<<<<<<<<<<<<<<<<<<<<<<<<<<

#ifndef callchk         >>>>>>>>>>>>>>>>>>>>>>>>>>>
# define callchk(x)     allchk(x)
#endif                  <<<<<<<<<<<<<<<<<<<<<<<<<<<

#ifndef PUTCHAR         >>>>>>>>>>>>>>>>>>>>>>>>>>>
# define PUTCHAR(x)     putchar(x)
#endif                  <<<<<<<<<<<<<<<<<<<<<<<<<<<

        /* macros for doing double indexing */
# define R2PACK(x,y)     (0200*((x)+1)+y)
# define R2UPK1(x)       (((x)>>7)-1)
# define R2UPK2(x)       ((x)&0177)
# define R2TEST(x)       ((x)>=0200)
```

In the discussion that follows, a linearized notation for trees is used, so that `A` denotes a subtree consisting of single node of type `A`, while `A ( B , C )` denotes a subtree that has `A` as its root, and `B` and `C` as its left and right descendents, respectively, etc. The characters `*`, `++`, `-=` are the familiar symbols from the C language.

The expression trees passed from the first to the second pass of the Portable C compiler contain only the simplest of these modes in a "ready-made" form, namely:

- `REG` — Operand is a register. [PDP11 address mode is "register", or 0*n*.]
- `NAME` — Symbolic address of operand is given. [Mode is "relative", or 67.]
- `ICON` — Immediate constant (possibly an address). [Mode is "immediate", or 27.]

As will be seen later in Chapter Seven, the routine `oreg2` (1988) is invoked by `canon` (1307) to recognize certain subtrees, and to convert these into nodes of type `OREG` (register plus offset). The program design envisages four different styles of `OREG`, of which only two are relevant for the PDP11, namely:

- `OREG` — Register contains a pointer to the operand. [Mode is "register deferred", or 1*n*.]
- `OREG` — The sum of the register plus the word following the instruction defines the address of the operand. [Mode is "index", or 6*n*.]

Each of the remaining address modes is recognized and handled as a subtree with two or more nodes, right up till the time of code generation. For example:

- `* ( NAME )` — The absolute address of a pointer to operand is given. [Mode is 77.]
- `* ( ICON )` — This is the same as the previous case.
- `* ( OREG )` — The address of a pointer to the operand is given by the sum of the register and the word following the instruction. [Mode is "index deferred" or 7*n*.]
- `* ( ++ ( REG, ICON ) )` The register is a pointer to the operand. After the reference is made, the value of the register is incremented by the value of the constant. [Provided the increment is appropriate to the operand (i.e., one for a character, two for a word, etc.), this can be handled by "autoincrement" addressing, i.e. mode 2*n*.]
- `* ( -= (REG, ICON ) )` The register is decremented, and then used to point to the operand. Provided the decrement is the appropriate value, "autodecrement" addressing, i.e. mode 4*n*, can be used.]

In this program, there are many places where a class of subtrees of depths varying from one to three, and which represent PDP11 addressing modes, must be recognized and handled. (For example, see line 2183 and `shltype`, 4141.) The concept of "shape" serves to characterize such subtrees. The last two examples above are considered to have the shape `STARREG`, and the three before those, the shape `STARNM`. However, only a limited number of the possible shapes are explicitly recognized, and the shape is not stored explicitly with the subtree.

It seems to the present writer that life would be much easier in many parts of the compiler if `oreg2`, or some equivalent, could reduce *all* the subtrees that represent addressing modes to one single node type. That this has not been done seems to be the result of an implicit assumption in the original design, namely that the set of operators and the definition of `OREG` would be machine-independent and non-negotiable. Steve Johnson says that an alternative approach, which he prefers, would involve extending for each machine the set of "special shapes" that would be recognized to include such cases.

```
# include "mfile2"

int nerrors = 0;  /* number of errors */

	/* VARARGS1 */
uerror( s, a ) char *s; { /* nonfatal error message */
	/* the routine where is different for pass 1 and pass 2:
	/*  it tells where the error took place */

	++nerrors;
	where('u');
	fprintf( stderr, s, a );
	fprintf( stderr, "\n" );
	if( nerrors > 30 ) cerror( "too many errors");
	}
/* --------------------------------------------------- */

	/* VARARGS1 */
werror( s, a, b ) char *s; {  /* warning */
	where('w');
	fprintf( stderr, "warning: " );
	fprintf( stderr, s, a, b );
	fprintf( stderr, "\n" );
	}
/* --------------------------------------------------- */

	/* VARARGS1 */
cerror( s, a, b, c ) char *s; { /* compiler error: die */
	where('c');
	if( nerrors && nerrors <= 30 ){ /* give the compiler the
					   benefit of the doubt */
		fprintf(stderr,
			"cannot recover from earlier errors: goodbye!\n");
		}
	else {
		fprintf( stderr, "compiler error: " );
		fprintf( stderr, s, a, b, c );
		fprintf( stderr, "\n" );
		}
	EXIT(1);
	}

/* =================================================== */

NODE *NIL;  /* pointer which always has 0 in it */

NODE *lastfree;  /* pointer to last free node; (for allocator) */

tinit(){ /* initialize expression tree search */
	NODE *p;

	for( p=node; p<= &node[TREESZ-1]; ++p ) p->op = FREE;
	lastfree = node;
	}
/* --------------------------------------------------- */

# define TNEXT(p) (p== &node[TREESZ-1]?node:p+1)

NODE *
talloc(){
	NODE *p, *q;

	q = lastfree;
	for( p = TNEXT(q); p!=q; p= TNEXT(p))
		if( p->op ==FREE ) return(lastfree=p);
```

This file contains procedures which are used in both passes of the compiler. Since certain structures, notably for the tree nodes, are defined differently in the two passes, this file is compiled with the file `mfile1` for the first pass, and with `mfile2` for the second pass. The full source code includes two files, `comm1.c` and `comm2.c`, which "include" the `common` file and the appropriate "mfile", and which are used in conjunction with the first and second passes respectively. (Neither `comm1.c` nor `comm2.c` is listed here.)

3.1 Error Messages

The three procedures, `uerror` (0599), `werror` (0612) and `cerror` (0621), are used to provide error messages on the standard output file, with varying degrees of severity. Note that a call to `cerror` is made when the compiler diagnoses a situation that "cannot happen". When this occurs the compilation is aborted. (The comments "`/* VARARGS1 */`" and similar ones elsewhere are for the benefit of `lint`.)

3.2 Tree Nodes

The next four procedures are concerned with the maintenance of tree structures. There are two other procedures that are companions to these, namely `ncopy` (2891) and `tcopy` (2910). The latter have been included in `allo.c` rather than the present file, because they are not used in the first pass.

0642: `tinit` is used to initialize the free list of nodes, which it does by setting the `op` field for every tree node, in the array `node` (0510), to the value `FREE`. The pointer `lastfree` (0640) is initialized to point to the first element of `node`.

0653: `talloc` finds the next "free" node and returns a pointer to it. Free nodes are found by searching forward from the last node allocated (designated by `lastfree`), wrapping around when the end of the array is reached. Compilation is terminated by the call to `cerror` at line 0660, if the free list becomes exhausted.

0665: `tcheck` checks that in a situation where there are no errors, all nodes in the array have been properly freed. This is a test for compiler consistency. If the test is satisfied, the only use for the subsequent call on `tinit` (0642) will be to set `lastfree`. This routine could obviously be improved so that the check will be performed when the errors are not of recent origin, and by calling `tinit` only when checking was not performed.

0675: `tfree`, as may easily be guessed, frees the nodes of a tree or subtree. The technique is to use the procedure `walkf` (0688) to perform (line 0678) an endorder walk of the tree, performing `tfree1` (0682) at each node visited.

3.3 Tree Walks

3.3.1 `walkf` *(0688)* performs an endorder walk over the tree whose root node is passed as its first parameter, and applies the function which is passed as its second argument to each node visited. The endorder traversal implies visiting the left subtree (if any), then the right subtree (if any), and then visiting the node itself. This is the appropriate algorithm to use when a bottom-up processing of the tree is required.

```
	cerror( "out of tree space; simplify expression");
	/* NOTREACHED */
	}
/* ---------------------------------------------------- */

tcheck(){ /* ensure that all nodes have been freed */
	NODE *p;

	if( !nerrors )
		for( p=node; p<= &node[TREESZ-1]; ++p )
			if( p->op != FREE ) cerror( "wasted space: %o", p );
	tinit();
	}
/* ---------------------------------------------------- */

tfree( p )  NODE *p; {  /* free the tree p */
	extern tfree1();

	if( p->op != FREE ) walkf( p, tfree1 );
	}
/* ---------------------------------------------------- */

tfree1(p)  NODE *p; {
	if( p == 0 ) cerror( "freeing blank tree!");
	else p->op = FREE;
	}
/* ---------------------------------------------------- */

walkf( t, f ) register NODE *t;  int (*f)(); {
	register opty;

	opty = optype(t->op);

	if( opty != LTYPE ) walkf( t->left, f );
	if( opty == BITYPE ) walkf( t->right, f );
	(*f)( t );
	}
/* ---------------------------------------------------- */

fwalk( t, f, down ) register NODE *t; int (*f)(); {
	int down1, down2;

	more:
	down1 = down2 = 0;

	(*f)( t, down, &down1, &down2 );

	switch( optype( t->op ) ){

	case BITYPE:
		fwalk( t->left, f, down1 );
		t = t->right;
		down = down2;
		goto more;

	case UTYPE:
		t = t->left;
		down = down1;
		goto more;

		}
	}
/* ==================================================== */
```

LTYPE nodes are leaf-type, and have no descendents; UTYPE nodes are unary type, and have a left descendent only; and BITYPE nodes are binary type, and have both left and right descendents.

3.3.2 `fwalk` *(0699)* performs a preorder walk over the tree whose root node is passed as its first parameter, and applies the function which is its second parameter to each node visited. The traversal involves visiting the node itself, then the left subtree (if any), and then the right subtree (if any). This procedure could be implemented purely recursively (as with `walkf`) but since the root node does not have to be revisited, there is the possibility, exploited here, of replacing recursion by iteration for visiting the (left) subtree of a UNARY node, and the right subtree of a binary node.

0705: At first glance, the variables, `down`, `down1` and `down2`, are somewhat perplexing. Since the function `f`, the first parameter, does not call itself recursively, there is no direct way for an invocation of `f` to pass information to its "descendents", i.e., invocations of `f` applied to nodes which are descendents of the current node. The second parameter, `down`, is a value which was passed to `f` by its "parent". In turn, it can deposit with its real parent, an invocation of `fwalk`, two values which are to be passed later to to its left and right "descendents". These last two values are passed back via the pointer arguments, `down1` and `down2`, respectively.

3.4 The `dope` arrays

The array `indope` (0727) is initialized with the values on lines 0729 to 0807. Each element of `indope` is a structure of type `dopest`, which contains:

1. an operator number, i.e. a value which may appear in the `op` field of a tree node.
2. an eight character array name for the operator, which is used for diagnostic printing.
3. a bitmask, stored as an integer, defining attributes of the operator, especially its type (LTYPE or UTYPE or BITYPE).

A careful study of these values now will be useful for later reference. As noted earlier, the operators are divided into three major categories, characterized as LTYPE, UTYPE and BITYPE. Many of the operator types declared in `manifest` do not occur (or at least should not occur because they are not expected) in the expression trees handled by the second pass. These include LB, RB, LC, RC, TYPE and STREF.

The binary operators, in particular, are classified into various groups, and group membership is indicated by the setting of flags in the `dopeval` field. The flags themselves are defined on lines 0141 to 0153. The meanings of the flags are fairly clear from the ways they are used on lines 0729 to 0805. As noted earlier, there are a number of definitions given on lines 0156 to 0159 that may be used for testing some of the flags in a convenient fashion.

3.5 `mkdope` (0811)

The ordering of the elements of `indope` is somewhat haphazard, and, in particular, is not constrained to be ordered by operator type. Hence two additional arrays are introduced, `dope` (0724) and `opst` (0725), which are indexed by operator type, and which allow direct retrieval of the `dopeval` bit mask, and the eight character operator name, respectively. The procedure `mkdope` (0811) is responsible for initializing `dope` and `opst` at object time. `mkdope` is called by `p2init` (0890).

3.6 `tprint` (0821)

This procedure which is called only from `eprint` at line 1167, during diagnostic printing of the contents of a tree, is straight forward enough. The only point which would require some explanation is the name in which the initial 't' stands, not for "tree", but for "type". Moreover it stands not for "operator type" but "operand type".

```
int dope[ DSIZE ];
char *opst[DSIZE];

struct dopest { int dopeop; char opst[8]; int dopeval; } indope[] = {

    NAME,       "NAME",    LTYPE,
    STRING,     "STRING",  LTYPE,
    REG,        "REG",     LTYPE,
    OREG,       "OREG",    LTYPE,
    ICON,       "ICON",    LTYPE,
    FCON,       "FCON",    LTYPE,
    CCODES,     "CCODES",  LTYPE,
    TYPE,       "TYPE",    LTYPE,

    NOT,        "!",       UTYPE|LOGFLG,
    COMPL,      "~",       UTYPE,
    FORCE,      "FORCE",   UTYPE,
    INIT,       "INIT",    UTYPE,
    SCONV,      "SCONV",   UTYPE,
    PCONV,      "PCONV",   UTYPE,
    FLD,        "FLD",     UTYPE,
    GOTO,       "GOTO",    UTYPE,
    STARG,      "STARG",   UTYPE,

    UNARY MINUS,    "U-",       UTYPE,
    UNARY MUL,      "U*",       UTYPE,
    UNARY AND,      "U&",       UTYPE,
    UNARY CALL,     "UCALL",    UTYPE|CALLFLG,
    UNARY FORTCALL,"UFCALL",    UTYPE|CALLFLG,
    UNARY STCALL,   "USTCALL",  UTYPE|CALLFLG,

    PLUS,       "+",       BITYPE|FLOFLG|SIMPFLG|COMMFLG,
    ASG PLUS,   "+=",      BITYPE|ASGFLG|ASGOPFLG|FLOFLG|SIMPFLG|COMMFLG,
    MINUS,      "-",       BITYPE|FLOFLG|SIMPFLG,
    ASG MINUS,"-=",        BITYPE|FLOFLG|SIMPFLG|ASGFLG|ASGOPFLG,
    MUL,        "*",       BITYPE|FLOFLG|MULFLG,
    ASG MUL,    "*=",      BITYPE|FLOFLG|MULFLG|ASGFLG|ASGOPFLG,
    AND,        "&",       BITYPE|SIMPFLG|COMMFLG,
    ASG AND,    "&=",      BITYPE|SIMPFLG|COMMFLG|ASGFLG|ASGOPFLG,
    QUEST,      "?",       BITYPE,
    COLON,      ":",       BITYPE,
    ANDAND,     "&&",      BITYPE|LOGFLG,
    OROR,       "||",      BITYPE|LOGFLG,
    CM,         ",",       BITYPE,
    COMOP,      ",OP",     BITYPE,
    ASSIGN,     "=",       BITYPE|ASGFLG,
    DIV,        "/",       BITYPE|FLOFLG|MULFLG|DIVFLG,
    ASG DIV,    "/=",      BITYPE|FLOFLG|MULFLG|DIVFLG|ASGFLG|ASGOPFLG,
    MOD,        "%",       BITYPE|DIVFLG,
    ASG MOD,    "%=",      BITYPE|DIVFLG|ASGFLG|ASGOPFLG,
    LS,         "<<",      BITYPE|SHFFLG,
    ASG LS,     "<<=",     BITYPE|SHFFLG|ASGFLG|ASGOPFLG,
    RS,         ">>",      BITYPE|SHFFLG,
    ASG RS,     ">>=",     BITYPE|SHFFLG|ASGFLG|ASGOPFLG,
    OR,         "|",       BITYPE|COMMFLG|SIMPFLG,
    ASG OR,     "|=",      BITYPE|COMMFLG|SIMPFLG|ASGFLG|ASGOPFLG,
    ER,         "^",       BITYPE|COMMFLG|SIMPFLG,
    ASG ER,     "^=",      BITYPE|COMMFLG|SIMPFLG|ASGFLG|ASGOPFLG,
    INCR,       "++",      BITYPE|ASGFLG,
    DECR,       "--",      BITYPE|ASGFLG,
    STREF,      "->",      BITYPE,
    CALL,       "CALL",    BITYPE|CALLFLG,
    FORTCALL,   "FCALL",   BITYPE|CALLFLG,
    EQ,         "==",      BITYPE|LOGFLG,
    NE,         "!=",      BITYPE|LOGFLG,
    LE,         "<=",      BITYPE|LOGFLG,
```

This is an appropriate occasion to draw attention to the way that operand type information is stored in a variable of type `TWORD` (which is defined to be an unsigned integer on line 0241). Such a variable is actually a packed structure of one four bit integer which defines the basic operand type (symbolic names for the sixteen possible values are defined on lines 0163 through 0178) plus a set of two bit integers defining type modifiers (see lines 0181 to 0183).

```
    LT,        "<",      BITYPE|LOGFLG,
    GE,        ">",      BITYPE|LOGFLG,
    GT,        ">",      BITYPE|LOGFLG,
    UGT,       "UGT",    BITYPE|LOGFLG,
    UGE,       "UGE",    BITYPE|LOGFLG,
    ULT,       "ULT",    BITYPE|LOGFLG,
    ULE,       "ULE",    BITYPE|LOGFLG,
    ARS,       "A>>",    BITYPE,
    LB,        "[",      BITYPE,
    CBRANCH,   "CBRANCH",BITYPE,
    PMCONV,    "PMCONV", BITYPE,
    PVCONV,    "PVCONV", BITYPE,
    RETURN,    "RETURN", BITYPE|ASGFLG|ASGOPFLG,
    CAST,      "CAST",   BITYPE|ASGFLG|ASGOPFLG,
    STASG,     "STASG",  BITYPE|ASGFLG,
    STCALL,    "STCALL", BITYPE|CALLFLG,

-1,   0
};
/* ------------------------------------------------------ */

mkdope(){
	register struct dopest *q;

	for( q = indope; q->dopeop >= 0; ++q ){
		dope[q->dopeop] = q->dopeval;
		opst[q->dopeop] = q->opst;
		}
	}
/* ====================================================== */

tprint( t )  TWORD t; {
	/* output a nice description of the type of t */

	static char * tnames[] = {
		"undef",
		"farg",
		"char",
		"short",
		"int",
		"long",
		"float",
		"double",
		"strty",
		"unionty",
		"enumty",
		"moety",
		"uchar",
		"ushort",
		"unsigned",
		"ulong",
		"?", "?"
		};

	for(;; t = DECREF(t) ){

		if( ISPTR(t) ) printf( "PTR " );
		else if( ISFTN(t) ) printf( "FTN " );
		else if( ISARY(t) ) printf( "ARY " );
		else {
			printf( "%s", tnames[t] );
			return;
			}
		}
	}
/* ------------------------------------------------------ */
```

Chapter 4: The File "reader.c" Part One

In this chapter we can begin discussion of the major procedures comprising the second pass of the Portable C compiler. This is the longest file, and in many ways the most difficult. It contains the procedures that determine the grand strategy for code generation and for the whole second pass.

The discussion of this file has been divided into four parts. This chapter considers the following procedures:

1. `p2init`, as the name suggests, performs initialization.
2. `main` reads the input file and calls the shots.
3. `rdin` reads numbers from the input file.
4. `eread` reads expression trees from the input file.
5. `eprint` displays expression trees for diagnostic purposes.
6. `delay` tries to break the expression tree into more manageable parts.
7. `delay1` looks for calculations that can be done immediately before the main calculation.
8. `delay2` looks for calculations that can be put off till later.

4.1 Variables

1. `filename` (0860) is used for the name of the source code file that is being compiled. This name is passed from the first pass to the second pass for diagnostic purposes.
2. `ftnno` records the number (arbitrarily) assigned to the current function. When this number changes in the data received from the first pass, the second pass must perform certain "end-of-function" chores.
3. `lineno` is passed from the first pass for diagnostic purposes. It refers to a line in the source code.
4. `lflag` may be set from the command line invoking the program. When it is set, a comment line identifying each input code line is sent to the assembler output of the compiler (see line 1022).
5. Beginning at line 0866, there are a set of "debugging" flags, `?debug`, which may be set and which provoke various kinds of diagnostic output for checking the program's behavior.
6. `tmpoff`, `maxoff`, `baseoff` and `maxtemp` are all used in the management of the current procedure's stack frame. The first three of these measure offsets from the beginning of the stack frame.

Explanations of the remaining variables, `maxtreg`, `fregs`, `stotree`, `stocook` and `callflag`, are given later.

4.2 `p2init` (0890)

`p2init` is the section of initialization code that is executed in both the one and the two pass version of the Portable C compiler. It is called as the very first action of `main` (0961) in the version of the source code presented here.

```
# include "mfile2"

/*	some storage declarations */

char filename[100] = "";  /* the name of the file */
int ftnno:  /* number of current function */
int lineno;
int nrecur;
int lflag;

int edebug  = 0;
int odebug  = 0;
int rdebug  = 0;
int radebug = 0;
int sdebug  = 0;
int tdebug  = 0;
int udebug  = 0;
int xdebug  = 0;

OFFSZ tmpoff;  /* offset for first temporary,
                              in bits for current block */
OFFSZ maxoff:  /* maximum temporary offset over all blocks
                              in current ftn. in bits */
OFFSZ baseoff = 0;
OFFSZ maxtemp = 0;

int maxtreg;
int fregs;
NODE *stotree;
int stocook;
int callflag;

/* ================================================= */

p2init( argc, argv ) char *argv[];{
	/* set the values of the pass 2 arguments */

	register int c;
	register char *cp;
	register files;

	allo0();  /* free all regs */
	files = 0;

	for( c=1; c<argc; ++c ){
		if( *(cp=argv[c]) == '-' ){
			while( *++cp ){
				switch( *cp ){

				case 'X':  /* pass1 flags */
					while( *++cp ) { /* VOID */ }
					--cp;
					break;

				case 'l':  /* linenos */
					++lflag;
					break;

				case 'e':  /* expressions */
					++edebug;
					break;

				case 'o':  /* orders */
					++odebug;
					break;
```

0897: `allo0` (2458) initializes a number of variables that are used in the allocation of the cpu registers.

0900: Loop through the arguments passed to the program by its parent, looking to see which options have been requested, incrementing the associated flags, and looking also for explicit file names (if any).

0955: Call `mkdope` (0811) to initialize the arrays, especially `dope` (0724), which describe the different operator types.

0956: `setrew` (2112) scans the contents of the array `table` (which contains the templates for the machine orders). It initializes `rwtable` (2108) and the array `opptr` (2110), which define starting points for searching `table` when operator templates are being matched for a given operation.

4.3 `main` (0961)

In the distributed source code, `main` actually occurs as `mainp2`. This procedure, whose principal function is to read the intermediate file written by the first pass, is not needed in the one pass version of the Portable C compiler.

0968: The value returned by `p2init` indicates whether there are explicitly named input files, or whether input data should be obtained from the standard input.

0969: `tinit` (0642) initializes the freelist of tree nodes.

0973: Re-read the argument list, looking for a file name (if such is known to exist i.e. was reported by `p2init`), and use it to reopen the standard input file.

0978: There is a bug* in the code here. Replace `files` by `files++`.

0980: Begin reading the standard input which is organized as a set of lines of ascii characters. Each line is classified by its first character.

0981: Lines beginning with ')' get copied directly to the standard output (assembler code and directives, which were generated during the first pass of the compiler).

0989: Lines beginning with '[' define the beginning of a new block. In Fortran the concepts of block and subroutine coincide. In C, a procedure may consist of more than one block. The beginning of a new procedure or subroutine implies, at object time, extension of the stack and adjustment of the stack pointer. The code for procedure prologues is generated in the first pass and does not concern the second pass, except in one respect: The stack pointer is advanced by an amount which is a symbolic constant representing the maximum growth of the stack frame during the procedure. The value of this constant is accumulated as `maxoff`. (See also the comments for lines 0997 and 1012 below.)

0990: `rdin` (1055) reads an optional minus sign plus a string of numeric characters from the input, and interprets them as a number in the base passed as an argument.

0994: The line should contain exactly three numbers, in base 10:

1. a function number.

---

* Pointed out by Lee Benoy, who never got around to fixing it.

```
				case 'r':  /* register allocation */
					++rdebug;
					break;

				case 'a':  /* rallo */
					++radebug;
					break;

				case 't':  /* ttype calls */
					++tdebug;
					break;

				case 's':  /* shapes */
					++sdebug;
					break;

				/* Sethi-Ullman testing (machine dependent) */
				case 'u':
					++udebug;
					break;

				/* general machine-dependent debugging flag */
				case 'x':
					++xdebug;
					break;

				default:
					cerror( "bad option: %c", *cp );
					}
				}
			}
		else files = 1;  /* assumed to be a filename */
		}
	mkdope();
	setrew();
	return( files );
	}
/* ----------------------------------------------- */

main( argc, argv ) char *argv[]; {
	register files;
	register temp;
	register c;
	register char *cp;
	register NODE *p;

	files = p2init( argc, argv );
	tinit();

	reread:

	if( files ){
		while( files < argc && argv[files][0] == '-' ) {
			++files;
			}
		if( files > argc ) return( nerrors );
		freopen( argv[files], "r", stdin );
		}
	while( (c=getchar()) > 0 ) switch( c ){
	case ')':
		/* copy line unchanged */
		while( (c=getchar()) > 0 ){
			PUTCHAR(c);
			if( c == '\n' ) break;
			}
		continue;
```

2. an offset, which defines where in the stack allocation of temporaries can begin, i.e. after the area allocated to automatic variables.
3. `maxtreg`, i.e. a statement of the maximum number of temporary registers available. (This can vary from block to block and depends on the number of register variables allocated.)

0997: If the function number has changed, re-initialize `maxoff`, `maxtemp` and `ftnno`. These are used as follows:

1. `maxoff` keeps track of the maximum value of `tmpoff` and `baseoff` over all expressions in a single function.
2. `maxtemp` keeps track of the number of temporary locations allocated, but is not otherwise used (at least in the PDP11 version).

1007: `setregs` is the first machine dependent routine that we encounter. Its principal function is to calculate `fregs`, the number of available type A scratch registers. This value, which is calculated afresh for each block, is determined as the larger of `maxtreg + 1` and `MINRVAR`, and for the PDP11, is never greater than four. For testing purposes, the value of `fregs` may be limited to a particular value by use of the "x" program debugging flag. `setregs` also updates the array `rstatus` at the beginning of each block to reflect the temporary or otherwise status of each register.

1010: Lines beginning with ']' denote the end of a block.

1011: `SETOFF` (0225) rounds the value of `maxoff` up to an even multiple of `ALSTACK`, which defines the preferred alignment boundary for stack entries. (On the PDP11, the value of `ALSTACK` (0266) is 16 (bits), implying alignment to a word boundary.

1012: `eobl2` (3755) is a machine dependent routine which performs "end of block" actions. For the PDP11 this consists of determining the maximum extent of stack growth for the block, and issuing an assembler directive. It will also define the variable `fltused` ("float used") for the assembler, if a floating point operation has been compiled. (This will subsequently influence the loading of certain library routines with the compiled program.)

1018: Lines beginning with a period define an expression tree for which code is to be generated.

1019: Read the source file line number.

1020: Read the source file name into the array `filename`.

1022: If `lflag` is set (which would have occurred in `p2init`), call `lineid` (3770) to display the line number and file name in the output file. Since this will be a comment in the assembler file, `lineid` is regarded as a machine dependent routine.

1026: `eread` (1089) is called to read in the details of the expression tree, and to recreate the expression tree in internal binary form.

This routine is somewhat time-consuming, and is avoided entirely in the one-pass version of the compiler. In situations where the two-pass version must be used, a worthwhile improvement in execution efficiency may be obtained by changing the mode of storage of expression trees from the current ascii form.

```
      case '[':   /* beginning of a block */
            temp = rdin(10);  /* ftnno */
            /* autooff for block gives max offset of autos in block */
            tmpoff = baseoff = rdin(10);
            maxtreg = rdin(10);
            if( getchar() != '\n' )
                  cerror( "intermediate file format error");

            if( temp != ftnno ){ /* beginning of function */
                  maxoff = baseoff;
                  ftnno = temp;
                  maxtemp = 0;
                  }
            else {
                  if( baseoff > maxoff ) maxoff = baseoff;
                  /* maxoff at end of ftn is max of autos and temps
                     over all blocks in the function */
                  }
            setregs();
            continue;

      case ']':   /* end of block */
            SETOFF( maxoff, ALSTACK );
            eobl2();
            while( (c=getchar()) != '\n' ){
                  if(c <= 0) cerror("intermediate file format eof");
                  }
            continue;

      case '.':   /* compile code for an expression */
            lineno = rdin( 10 );
            for( cp=filename; (*cp=getchar()) != '\n'; ++cp ) { }
            *cp = '\0';
            if( lflag ) lineid( lineno, filename );

            /* expression at top level reuses temps */
            tmpoff = baseoff;
            p = eread();

            if( edebug ) fwalk( p, eprint, 0 );

# ifdef MYREADER        >>>>>>>>>>>>>>>>>>>>>>>>
            MYREADER(p);  /* do your own laundering of the input */
# endif                 <<<<<<<<<<<<<<<<<<<<<<<<

            nrecur = 0;
            delay( p ); /* expression statement throws out results */
            reclaim( p, RNULL, 0 );

            allchk();
            tcheck();
            continue;

      default:
            cerror( "intermediate file format error" );

            }

      /* EOF */
      if( files ) goto reread;
      return(nerrors);
      }

/* ================================================== */
```

1028: For debugging purposes, call `eprint` (1134) at each node to print a display of the expression tree.

1031: The optional call on `myreader` at this point provides an opportunity for some machine dependent massaging of the expression tree.

For the PDP11, `myreader` (3926) (a) looks for "hard operations", namely multiply and divide operations involving long integers, and rewrites the tree so that these operations are replaced by calls on library procedures; (b) rewrites expressions involving the operator AND, so that its meaning is effectively changed to that of the PDP11 "bic" instruction (since this operation is non-commutative, care has to be exercised later to avoid re-ordering the left and right subtrees); and (c) resets `toff` to zero.

1035: This innocent looking procedure call to `delay` (1183) initiates code generation for the expression tree. By the time it is finished, there should be nothing much left ...

1036: for `reclaim` (2677) to salvage, and ...

1038: for `allchk` (2479) and `tcheck` (0665) to check.

4.4 `rdin` (1055)

`rdin` is a routine for reading in integer numbers in ascii format, without generating overflows at the extreme end of the range. Numbers may begin with zero or more minus signs, and must terminate with a tab character. The number base is provided as an argument. With only one exception, (line 1106), `base` is always 10 when `rdin` is called.

4.5 `eread` (1089)

This procedure is called by `main` at line 1026 to read the input file and build the expression tree in internal form.

1098: Read the operator value and assign it to the `op` field of a newly acquired node.

1102: The operator type, `LTYPE` (leaf), `UTYPE` (unary operator), or `BITYPE` (binary operator), determines subsequent actions at this level. `UTYPE` nodes always (by convention) have a left subtree, but no right subtree.

1103: For `LTYPE` nodes, get a value for `lval`.

1104: For `LTYPE` and `UTYPE` nodes, get a value for `rval`.

1106: Read in the operand type information, i.e. a numeric value which which will be interpreted as one of CHAR, `SHORT`, `INT`, `LONG`, etc.

1107: Initialize the `rall` field to indicate no preference for locating the result of the operation. This field may subsequently be changed by `rallo` for particular operations, to indicate a preference for, or insistence upon, a particular register.

1109: If the operation involves structures, read and save values defining the structure size and the required storage alignment boundary parameter. These four operations: structure assignment, structure argument, and call to a function, with or without arguments, which returns a structure as a result, are either `UTYPE` or `BITYPE`. Note also that `stsize` and `stalign` occupy space that is otherwise assigned to the `name` array in the tree node.

```
CONSZ
rdin( base ){
	register sign, c;
	CONSZ val;

	sign = 1;
	val = 0;

	while( (c=getchar()) > 0 ) {
		if( c == '-' ){
			if( val != 0 ) cerror( "illegal -");
			sign = -sign;
			continue;
			}
		if( c == '\t' ) break;
		if( c>='0' && c<='9' ) {
			val *= base;
			if( sign > 0 )
				val += c-'0';
			else
				val -= c-'0';
			continue;
			}
		cerror("illegal character `%c' on intermediate file", c);
		break;
		}

	if( c <= 0 ) {
		cerror( "unexpected EOF");
		}
	return( val );
	}
/* --------------------------------------------------- */

NODE *
eread(){

	/* call eread recursively to get subtrees, if any */

	register NODE *p;
	register i, c;
	register char *pc;
	register j;

	i = rdin( 10 );
	p = talloc();
	p->op = i;

	i = optype(i);
	if( i == LTYPE ) p->lval = rdin( 10 );
	if( i != BITYPE ) p->rval = rdin( 10 );

	p->type = rdin(8 );
	p->rall = NOPREF;  /* register allocation information */

	if( p->op == STASG || p->op == STARG ||
		    p->op == STCALL || p->op == UNARY STCALL ){
		p->stsize = (rdin( 10 ) + (SZCHAR-1) )/SZCHAR;
		p->stalign = rdin(10) / SZCHAR;
		if( getchar() != '\n' ) cerror( "illegal \n" );
		}
```

1116: If the operator is a register, increment the register's "busy" count.

1119: Read in the name, and store up to NCHNAM (eight) characters.

1122: Add a null character at the end of the name, if it is less than NCHNAM characters long.

1127: For UTYPE and BITYPE operators, read the left subtree.

1128: For BITYPE operators, read the right subtree.

Note that no nodes representing labels (form "C", line 0487) are expected by this routine.

4.6 `eprint` (1134)

This procedure is used to provide a diagnostic display of the expression tree during debugging. It is referenced from several locations, but always as an argument to `fwalk` (0699), viz.

```
fwalk ( p, eprint, 0 );
```

A proper understanding of this procedure is not necessary for our immediate purpose, but is otherwise instructive, and it does cast some light on the type of information which may be stored in the tree nodes, e.g.

1. REG nodes (which are of type LTYPE, i.e. a leaf) have the associated register number stored as `rval`.
2. ICON, NAME and OREG nodes have an associated address part which is stored as `lval`. In the case of OREG nodes, a register number is also stored as `rval`.
3. `rall` can contain one of the patterns: NOPREF or PREF plus a register number, or MUSTDO plus a register number.

From lines 1137 to 1141, it will be seen that the equivalent of `down` times four blanks are emitted at the beginning of each line and that the value passed to the "descendent" is increased by one at each stage. (See the discussion of `fwalk` (0699) in the previous chapter.) Thus there are provided different levels of indentation for each level of the tree.

4.7 `delay` (1183)

This routine looks for ways of breaking the expression tree into (smaller) subtrees, that can be handled more expeditiously.

1191: Call `delay1` repeatedly to break off the left subtrees of any "visible" comma-operators, and process them immediately. Finish when they are all done. When a comma-operator occurs within an expression, it implies that the calculation represented by the left subtree should be carried out for side effects only, and the value returned for the whole tree should be the value obtained from evaluating the right subtree. In this situation, a comma-operator is considered invisible if it occurs in part of the expression which is not always evaluated, i.e. that is part of the right subtree of an operator such as ANDAND or OROR.

1195: Call `delay2` to find right subtrees which can be broken off for processing later. (References to these are accumulated in the array `deltrees` (1180).)

1196: Call `codgen` (1281) to process the remaining trunk of the original tree.

1198: Call `codgen` to process all the subtrees split off by `delay2`.

4.8 `delay1` (1202)

This procedure performs a (possibly abbreviated) preorder traversal of the tree, looking for visible COMOPs (comma-operators). As noted already, such an operator will be regarded as not visible, if it is part of the right subtree for a conditional operator, i.e. a subtree for which the

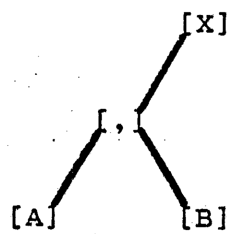
```
        else {    /* usual case */
              if( p->op == REG )
                    /* non usually, but sometimes justified */
                    rbusy( p->rval, p->type );
              for( pc=p->name,j=0; ( c = getchar() ) != '\n'; ++j ){
                    if( j < NCHNAM ) *pc++ = c;
                    }
              if( j < NCHNAM ) *pc = '\0';
              }

        /* now, recursively read descendents, if any */

        if( i != LTYPE ) p->left = eread();
        if( i == BITYPE ) p->right = eread();

        return( p );
        }
/* ------------------------------------------------------ */

eprint( p, down, a, b ) NODE *p; int *a, *b; {

        *a = *b = down+1;
        while( down >= 2 ){
              printf( "\t" );
              down -= 2;
              }
        if( down-- ) printf( "    " );

        printf( "%o) %s", p, opst[p->op] );
        switch( p->op ) { /* special cases */

        case REG:
              printf( " %s", rnames[p->rval] );
              break;

        case ICON:
        case NAME:
        case OREG:
              printf( " " );
              adrput( p );
              break;

        case STCALL:
        case UNARY STCALL:
        case STARG:
        case STASG:
              printf( " size=%d", p->stsize );
              printf( " align=%d", p->stalign );
              break;
              }

        printf( ", " );
        tprint( p->type );
        printf( ", " );
        if( p->rall == NOPREF ) printf( "NOPREF" );
        else {
              if( p->rall & MUSTDO ) printf( "MUSTDO " );
              else printf( "PREF " );
              printf( "%s", rnames[p->rall&~MUSTDO]);
              }
        printf( ", SU= %d\n", p->su );
        }

/* ====================================================== */
```

corresponding subexpression will not always be evaluated during the evaluation of the whole expression. For example, if the expression

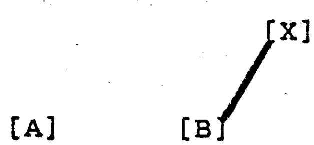
A && B

is evaluated in a conditional context, and the value of A is found to be false, then it is not necessary to evaluate B also in order to determine the value of the whole expression. In fact in the C language, it is expressly required that B should not be evaluated in this situation, and A may be a test to determine whether the evaluation of B will not cause an object time error.

In terms of tree operations, the tree



is to be transformed into two separate trees



with the leftmost of these being processed immediately via the recursive call on line 1219.

In the process of splitting the tree, one node is freed. Formally, this is the node containing the COMOP. However, since there may be several references to this node recorded in difficult to find places, for example, as actual arguments to procedures, it turns out to be convenient, if not absolutely essential, to free the node which was the root of the right subtree after its contents have been copied (by `ncopy` (2891)) onto the node which was formerly the COMOP.

1207: Leaf nodes are obviously of no interest. Return a zero value.

1208: Unary nodes are not COMOPs. Look down the left subtree (the only possibility).

1210: With only binary nodes left to deal with, if the operator is ...

1212: QUEST or ANDAND or OROR, do not look for COMOPs in the right subtree (yet).

1218: At last!

1219: Call `delay` (1183) recursively to handle the left subtree. Upon return from this procedure call, the left subtree will be completely reduced in the sense that code for all the computations represented by that subtree will have been generated. Since the COMOP takes its value from the right subtree, there is no need to investigate the value, if any, calculated by the left subtree.

1221: Re-write the subtree. `ncopy` (2891) copies the contents of the node referenced by its second argument onto the node referenced by its first argument. (i.e. the "lvalue" is on the left.)

```
NODE *deltrees[DELAYS];
int  deli;

delay( p ) register NODE *p; {
      /* look in all legal places for COMOP's, ++ & -- ops to delay */
      /* note: don't delay ++ and -- within calls or things like
      /* getchar (in their macro forms) will start behaving strangely
      */
      register i;

      /* look for visible COMOPS, and rewrite repeatedly */
      while( delay1( p ) ) { /* VOID */ }

      /* look for visible, delayable ++ and -- */
      deli = 0;
      delay2( p );
      codgen( p, FOREFF );  /* do what is left */
      /* do the rest */
      for( i = 0; i<deli; ++i ) codgen( deltrees[i], FOREFF );
      }
/* ------------------------------------------------ */

delay1( p ) register NODE *p; {  /* look for COMOPS */
      register o, ty;

      o = p->op;
      ty = optype( o );
      if( ty == LTYPE ) return( 0 );
      else if( ty == UTYPE ) return( delay1( p->left ) );

      switch( o ){

      case QUEST:
      case ANDAND:
      case OROR:
            /* don't look on RHS */
            return( delay1(p->left ) );

      case COMOP:  /* the meat of the routine */
            delay( p->left );  /* completely evaluate the LHS */
            /* rewrite the COMOP */
            { register NODE *q;
                  q = p->right;
                  ncopy( p, p->right );
                  q->op = FREE;
                  }
            return( 1 );
            }

      return( delay1(p->left) || delay1(p->right ) );
      }
/* ------------------------------------------------ */

delay2( p ) register NODE *p; {

      /* look for delayable ++ and -- operators */

      register o, ty;
      o = p->op;
      ty = optype( o );

```

1226: Return a non-zero (true) value, which will be passed back eventually to `delay` at line 1191.

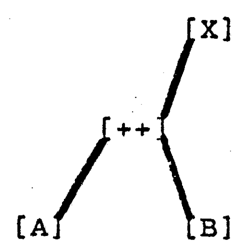
1229: None of the above-mentioned cases has occurred, so recursively search the left subtree, and if nothing interesting happens, do the right subtree also.

`delay` calls `delay1` repeatedly until no further changes are observed. After the tree has been broken up, and re-written by `delay1`, it is apparently necessary to return to the root of the whole tree. If the tree contained several COMOPs, not all of these may have been found upon the first try, and after the tree has been rewritten, the remaining COMOPs may appear at any node, including the root itself. (Consider the case of a tree in which COMOPs are cascaded to the right.)

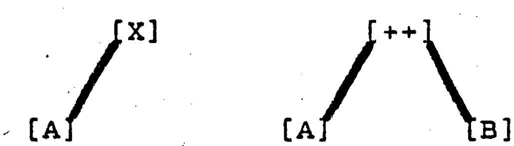
4.9 `delay2` (1233)

This procedure performs a preorder traversal of the tree, looking for visible `INCR` and `DECR` operators. These operators, which correspond to the postfix versions of ++ and -- in the C language are binary operators whose value is the value of the left operand, but which have the side-effect of changing the value of the operand. (The prefix versions of these operators are transformed into `ASG PLUS` and `ASG MINUS` operators respectively during the first pass of the compiler.) The ++ and -- operators constitute one of the more novel and innovative features of the C language. They also lead to some of the more intricate and complex parts of the C language compiler. Since they provide the application programmer with what turns out to be a two-edged sword, all in all, they have to be regarded as a mixed blessing.

If the current subtree looks like



it is to be replaced by a reduced subtree plus an extra tree



The extra tree is generated by making an entire copy of the original subtree (using `tcopy` (2910)), copying the left node, labeled 'A' in the diagram, onto the root node, and finally abandoning the two nodes labeled 'A' and 'B'.

1261: `deltest` (2947) is a machine dependent routine which determines under which conditions it is reasonable to delay the operation. For the PDP11, the decision to delay is taken if the left tree represents an addressable variable ("lvalue"), and the incrementation can not be achieved using autoincrement addressing.

1266: The node labeled 'B' will always be a leaf (i.e. a constant) from the way the expression tree was originally calculated.

Code generation for the extra tree will be delayed until after the main tree has been completely reduced. (see line 1198). Note also that if there is no node labeled 'X', i.e. the root node is the "++" node, `delay2` will still operate to create two trees, the first of which will be trivial, and will generate no code.

```
	switch( o ){

	case NOT:
	case QUEST:
	case ANDAND:
	case OROR:
	case CALL:
	case UNARY CALL:
	case STCALL:
	case UNARY STCALL:
	case FORTCALL:
	case UNARY FORTCALL:
	case COMOP:
	case CBRANCH:
		/* for the moment, don't delay past a conditional
		/* context, or inside of a call */
		return;

	case INCR:
	case DECR:
		if( deltest( p ) ){
			if( deli < DELAYS ){
				register NODE *q;
				deltrees[deli++] = tcopy(p);
				q = p->left;
				p->right->op = FREE;  /* zap constant */
				ncopy( p, q );
				q->op = FREE;
				return;
				}
			}

		}

	if( ty == BITYPE ) delay2( p->right );
	if( ty != LTYPE ) delay2( p->left );
	}

/* ================================================ */

codgen( p, cookie ) NODE *p; {

	/* generate the code for p;
	   order may call codgen recursively */
	/* cookie is used to describe the context */

	for(;;){
		/* create OREG from * if possible and do sucomp */
		canon(p);
		stotree = NIL;
		if( edebug ){
			printf( "store called on:\n" );
			fwalk( p, eprint, 0 );
			}
		store(p);
		if( stotree==NIL ) break;

		/* because it's minimal, can do w.o. stores */

		order( stotree, stocook );
		}
	order( p, cookie );

	}
/* ================================================ */
```

This chapter introduces the second set of procedures from the file `reader.c`. These are

1. `codgen` which attempts to generate code for a given subtree, for a specified effect.
2. `canon` which tidies up the tree, and calls `sucomp` to recalculate the Sethi-Ullman numbers.
3. `store` which looks for situations where temporary results must be placed outside the temporary registers.
4. `stoarg` which is a modified version of `store` for function arguments.
5. `markcall` which searches subtrees looking for "call" operators.
6. `constore`, which is a reduced version of `store`, is used to preserve the left-to-right evaluation of logical expressions.
7. `prcook` which is used for diagnostic printing.
8. `rcount`which keeps an iteration count, and terminates the compilation if things appear to be getting out of hand.

5.1 `codgen` (1281)

This procedure is called by `delay` (1196, 1198), after all obvious tree-lopping has been performed. It is also called (indirect recursion) by `order` and `cbranch`.

1287: Loop repeatedly, transforming the tree (via `canon` (1307)), and then calling `store` (1325) to look for a subtree whose value (i.e. the value which will be calculated at object time) needs to be stored in a temporary location outside the processor registers, in the run-time stack.

As long as such subtrees can be found, the call to `order` (1524) at line 1300 should generate segments of code, and simplify the tree, until finally the tree is simple enough to be handled directly by the final call to `order` at line 1302.

The main strategy of the second pass is laid bare at this point: as long as the current tree represents a calculation which is too complex to be carried out entirely within the processor's high speed, readily addressable registers, use the procedure `store` to identify a subtree which represents a calculation that can be so conducted and arrange to have the result of this calculation stored outside the registers, i.e. in a temporary core location. Use `order` to generate the code for this subtree. Simplify the main tree to take account of this, and try again, until the whole tree is computable.

In theory, there is a clear division of labor, with `store` making the strategic decisions and `order` doing the hack work. In practice, things are a little more complex. Due to the way conditional expressions are handled, `order` in fact calls `codgen` recursively in certain situations.

5.2 `canon` (1307)

This procedure is called principally by `codgen` and `order`, but also by `myreader`, `genargs` and `setasop`. Its function is to tidy up the expression tree in the following respects:

```
canon(p) NODE *p; {
	/* put p in canonical form */
	int oreg2(), sucomp();

# ifndef FIELDOPS        >>>>>>>>>>>>>>>>>>>>>
	int ffld();
	fwalk( p, ffld, 0 ); /* look for field operators */
# endif                  <<<<<<<<<<<<<<<<<<<<<
	walkf( p, oreg2 );  /* look for and create OREG nodes */
# ifdef MYCANON          >>>>>>>>>>>>>>>>>>>>>
	MYCANON(p);  /* your own canonicalization routine(s) */
# endif                  <<<<<<<<<<<<<<<<<<<<<
	walkf( p, sucomp );  /* do the Sethi-Ullman computation */

	}

/* ============================================== */

store( p ) register NODE *p; {

	/* find a subtree of p which should be stored */

	register o, ty;

	o = p->op;
	ty = optype(o);

	if( ty == LTYPE ) return;

	switch( o ){

	case UNARY CALL:
	case UNARY FORTCALL:
	case UNARY STCALL:
		++callflag;
		break;

	case UNARY MUL:
		if( asgop(p->left->op) ) stoasg( p->left, UNARY MUL );
		break;

	case CALL:
	case FORTCALL:
	case STCALL:
		store( p->left );
		stoarg( p->right, o );
		++callflag;
		return;

	case COMOP:
		markcall( p->right );
		if( p->right->su > fregs ) SETSTO( p, INTEMP );
		store( p->left );
		return;

	case ANDAND:
	case OROR:
	case QUEST:
		markcall( p->right );
		if( p->right->su > fregs ) SETSTO( p, INTEMP );
	case CBRANCH:   /* to prevent complicated expressions
			on the LHS from being stored */
	case NOT:
		constore( p->left );
		return;

		}

	if( ty == UTYPE ){
		store( p->left );
		return;
		}
```

1. If the cpu has no hardware for extracting subfields from words in storage directly, simulate the desired operation using a combination of shift and masking operations.
2. Replace explicit address calculations by implicit calculations that can be performed by the memory addressing hardware.
3. Perform any other transformations that will take advantage of the features of a particular machine (not used for the PDP11).
4. Finally, perform the "Sethi-Ullman" calculation (see Chapter Eleven) to determine the resource requirements (measured in numbers of type A cpu temporary registers) to carry out the calculation represented by each subtree.

Since `canon` is called quite frequently, since tree walking is a relatively expensive exercise, and since relatively few C expressions contain any reference to values stored in bit fields, the call here on `ffld` must be considered relatively expensive. Fields cannot be disposed of once and for all, because only field extractions, not field insertions may be handled easily. The rewriting of trees in mid-stream, e.g. for an assignment operator, may cause a field extraction to appear in the tree at some intermediate stage, but only if the tree contained a field insertion in the first place. If it were known that the tree contained no field operations, the common situation, then `ffld` need never be called.

If a bit mask were defined for each operator type, and the union of the masks for all operators present in the tree was created when the tree was built or reconstructed, then it would be possible to answer relatively inexpensively a number of simple questions such as "are there any field operators in the tree?". Not only could unnecessary tree walks looking for `FFLD` nodes be avoided, but `delay1` need not be called if there are no `COMOP` nodes, nor `delay2`, if there are no `INCR` or `DECR` nodes, etc.

Note that whereas `oreg2` and `sucomp` are constrained to walk the tree in endorder, `ffld` is not so constrained, and hence can use the faster preorder walk.

5.3 `store` (1325)

The principal call on this procedure occurs at line 1295 in `codgen`. It is also called recursively, directly at lines 1351, 1359, 1376, 1387, and 1388, and indirectly via `constore` (line 1468) and `stoarg` (lines 1409, 1414).

The basic idea behind `store` is simple enough: a pre-order walk, from right to left, is performed over the tree. If the node represents a "call" operation, `callflag` is incremented; if the SU number for the node is greater than the number of free registers, then the node is remembered as `stotree`, and an associated goal is remembered as `stocook`. These latter are set via the macro `SETSTO` (0532). Note that `SETSTO` will erase any values stored earlier, so that only the most recent values are saved. `store` is called iteratively until nothing further is to be done.

One may wonder whether a different method of traversing the tree would allow `store` to terminate the first time `SETSTO` is invoked. However the need to treat specially the right subtrees of conditional operators seems to forbid this.

`callflag` is zeroed only by `stoarg` (1392), just before the latter calls `store` (line 1408). The value of `callflag` is tested when nested calls must be avoided, i.e. a procedure call must not be invoked during the calculation of arguments for another procedure call. This can occur with machines that do not implement a hardware stack.

The basic structure of `store` is perturbed by a number of special situations:

1342: The `break` here is equivalent to a transfer to line 1376.

1345: For the PDP11, the only effect of `stoasg` (2960) is to return a value. See the discussion of the next section.

```

	if( asgop( p->right->op ) ) stoasg( p->right, o );

	if( p->su>fregs ){ /* must store */
		mkadrs( p );  /* set up stotree and stocook to subtree
				 that must be stored */
		}

	store( p->right );
	store( p->left );
	}
/* ------------------------------------------------------- */

stoarg( p, calltype ) register NODE *p; {
	/* arrange to store the args */

	if( p->op == CM ){
		stoarg( p->left, calltype );
		p = p->right ;
		}
	if( calltype == CALL ){
		STOARG(p);
		}
	else if( calltype == STCALL ){
		STOSTARG(p);
		}
	else {
		STOFARG(p);
		}
	callflag = 0;
	store(p);
# ifndef NESTCALLS	>>>>>>>>>>>>>>>>>>>>>>
	if( callflag ){
		/* prevent two calls from being active at once  */
		SETSTO(p,INTEMP);
		store(p); /* do again to preserve bottom up nature...  */
		}
# endif			<<<<<<<<<<<<<<<<<<<<<<
	}
/* ------------------------------------------------------- */

markcall( p ) register NODE *p; {
	/* mark off calls below the current node */

	again:
	switch( p->op ){

	case UNARY CALL:
	case UNARY STCALL:
	case UNARY FORTCALL:
	case CALL:
	case STCALL:
	case FORTCALL:
		++callflag;
		return;

		}

	switch( optype( p->op ) ){

	case BITYPE:
		markcall( p->right );
	case UTYPE:
		p = p->left;
		/* eliminate recursion (aren't I clever...) */
		goto again;
	case LTYPE:
		return;
		}
	}
/* ------------------------------------------------------- */

```

1352: The right subtree contains the argument list.

1357: `markcall` looks to see if there are any call operators in the subtree. It can be regarded as a restricted version of `store`.

1358: The right subtree of the COMOP is too complex for calculation entirely within the processor registers. Remember the place, and explore the left subtree.

1365: `markcall` (1420) explores the right subtree to check if there exist any "call operators" (not important for the PDP11).

1366: Evaluation of the right subtree can use all the available registers, so attempt to get the value of the whole conditional expression in to a temporary stack location.

1370: `constore` (1451) will follow the leftmost path of the subtree, without executing SETSTO, until a non-conditional operation is encountered.

1380: See the comments above for line 1345.

1383: `mkadrs` (2968) is called when it is known that a subtree is overloaded. Its job is to locate the most pressing subgoal, which should be satisfied. One way or the other, it invokes the macro SETSTO (0532), to set the values for the variables `stotree` (0884) and `stocook` (0885).

Since `store` is called recursively immediately following the call on `mkadrs`, and since SETSTO (0532) may be reinvoked during these recursive calls, it would seem possible and desirable to delay the calculation at line 1383 until after the call on line 1388, and then to invoke `mkadrs` only if `stotree` has not been reset.

5.4 `stoarg` (1392)

This procedure is called by `store` at line 1352.

Since for the PDP11, STOARG (0337), STOSTARG (0340), STOFARG (0339) and NESTCALLS (0352) are all defined with null values, the routine could be re-written simply as

```
if( p->op ==CM )
    stoarg( p->left, calltype );
callflag = 0;
store( p );
```

Since the value of `callflag` is only interrogated at line 1411, it is clear that the whole routine could in fact be replaced by the single statement

```
store( p );
```

This implies that for machines such as the PDP11 and VAX11/780, where nested calls are not a problem, the code in this area could be substantially revised.

1412: This comment is misleading, if taken literally, since the only possible side-effects of `store` are to set `callflag`, `stocook` and `stotree`. The intent here is, that when calls must not be nested, to do something in the situation where the call on `store` at line 1409 changed the value of `callflag`, but may or may not have changed the value of `stotree` (0884) and `stocook` (0885). If the values were changed, well and good, but if not, then line 1413 is what is required. This code comes under the heading of things that could be better said ...

5.5 `markcall` (1420)

As has already been noted, this procedure can be regarded as a stripped down version of `store`. As has been further noted, since its only side-effect is to change `callflag`, which is not of interest on machines such as the PDP11 and VAX11/780, it could be eliminated from the compiler for these machines.

```
constore( p ) register NODE *p; {

	/* store conditional expressions */
	/* the point is, avoid storing expressions in conditional
	   context, since the evaluation order is predetermined */

	switch( p->op ) {

	case ANDAND:
	case OROR:
	case QUEST:
		markcall( p->right );
	case NOT:
		constore( p->left );
		return;

		}
	store( p );
	}

/* ============================================== */

char *cnames[] = {
	"SANY",
	"SAREG",
	"STAREG",
	"SBREG",
	"STBREG",
	"SCC",
	"SNAME",
	"SCON",
	"SFLD",
	"SOREG",
	"STARNM",
	"STARREG",
	"INTEMP",
	"FORARG",
	"SWADD",
	0,
	};

prcook( cookie ){

	/* print a nice-looking description of cookie */

	int i, flag;

	if( cookie & SPECIAL ){
		if( cookie == SZERO ) printf( "SZERO" );
		else if( cookie == SONE ) printf( "SONE" );
		else if( cookie == SMONE ) printf( "SMONE" );
		else printf( "SPECIAL+%d", cookie & ~SPECIAL );
		return;
		}
	flag = 0;
	for( i=0; cnames[i]; ++i ){
		if( cookie & (1<<i) ){
			if( flag ) printf( "|" );
			++flag;
			printf( cnames[i] );
			}
		}
	}
/* ---------------------------------------------- */

rcount(){ /* count recursions */
	if( ++nrecur > NRECUR ){
	    cerror("expression causes compiler loop: try simplifying");
	    }
	}

/* ============================================== */
```

5.6 `constore` (1451)

This procedure is called by `store` at line 1370, when the operator of the tree node is found to be a conditional or a branch. The right subtrees of trees passed to `constore` are checked via `markcall` only for the presence of "call" operators. In cases where there are several conditional operators cascaded to the left, the use of `constore` on the second and subsequent nodes will ensure that any execution of `SETSTO` will be for the first such node only (see line 1366).

The special treatment accorded by `store` and `constore` to conditional operations, is to preserve the left to right evaluation of conditional expressions. `order`, when it comes to deal with such expressions, does not iterate or call itself recursively, which is its usual style of behavior. Instead, it calls `codgen` recursively (see line 1630, for example) to ensure that the order in which code will be generated will follow the C language rules.

5.7 `prcook` (1492)

Not much difficulty here. Diagnostic printing only.

5.8 `rcount` (1516)

This small procedure monitors the progress of the calculation, and provides an escape hatch in certain looping situations that could be endless. `nrecur` is reset to zero (line 1034) after each new expression tree is read in. `NRECUR` (0563) is defined to be ten times the number of tree nodes. `rcount` is called at the beginning of `order` (line 1537) and of `match` (line 2168).

Chapter 6: The File "reader.c" Part Three

This chapter is given over to the study of a single procedure, `order` (1524). This procedure is most probably the most challenging, as well as the longest, procedure of the program. Once you have mastered this, "the rest is downhill". Of course the task is not going to be exactly trivial, since this procedure is richly connected to many others. What makes it so formidable at first sight are:

1. its sheer length;
2. the number of machine dependent procedures that have been exorcised and moved into the machine-dependent file `order.c`, namely:

`setasg`	`setbin`	`setstr`
`setasop`	`setincr`	

3. the seemingly endless set of special cases (and how can you be sure that nothing has been missed?)
4. the many invocations of other procedures: (`order` makes in fact 55 separate procedure calls, to 30 different procedures.)
5. the extensive use of recursion, including six direct calls to itself, and numerous possibilities for indirect calls via procedures such as

`codgen`	`offstar`	`setbin`
`genargs`	`setasg`	`setstr`
`gencall`	`setasop`	

   The principal call to `order` occurs in `codgen` (lines 1300, 1302), after `store` (1325) has found a subtree that it thinks represents a calculation that can be conducted entirely within the registers of the cpu, i.e. without disturbing any registers that may have been temporarily reserved, and without requiring any intermediate results to be stored in the run-time stack.

6.1 Comparison with `codgen`

`codgen` and `order` are called with similar arguments, and have very similar intended functions: to generate code for a particular subtree (whose root is the first parameter), and to achieve a specified goal (the "cookie" or second argument). The difference between the two is ostensibly that, when `codgen` is called, it will not be clear whether any intermediate results will need to be stored in the run-time stack, whereas when `order` is called the latter doubt has been removed. However truth is stranger than fiction, and because `store` can only take a limited cognizance of conditional operators, in view of the ordering rules for conditional expression evaluation, `order` must be constrained in its behavior. Hence there are occasions where, instead of calling itself recursively, it must in fact go back to `codgen` to handle a particular subtree. Providing this is always a proper subtree, the process must eventually converge.

6.2 Strategy

The overall strategy used by `order` is broadly as follows:

1. it takes the subtree given as the first argument, and sees if it is matched by any of the templates provided in `table`.
2. if not, it perturbs the tree, either by explicitly rewriting it in some way, and/or by calling itself recursively to handle some subtree.

3. after the tree has been rewritten in some way, it returns to the beginning and tries again.

In the best of all possible worlds, `order` should be able to achieve its task with a "bottom up" strategy realized via an endorder traversal of the tree. In the method realized here, the strategy is neither purely "bottom up" nor "top down", but a hybrid of these. If the tree passed to `order` is clearly too complex to be translated into a single instruction (or group of instructions matched by a single template), then an attempt is made to reorganize the root portion of the tree in certain special cases, and the problem is tackled again from the beginning.

Before long, via one or more recursive calls, `order` will be dealing with subtrees that can be matched, and which after the corresponding code has been emitted, can be shrunk down to a single node, thus pruning the original tree. With the tree pruned, control can return to the higher level for yet another high level iteration, and so on. Note that not every subtree may be distinguished by its own call to `order`, since the procedure may reach two or more levels down into the current tree before calling itself recursively.

In calling itself recursively, not only does `order` skip some nodes, but it takes a fairly optimistic view of what can be achieved, and for the most part requests that intermediate results be left in a register. However, for certain types of nodes, if it looks like the going may get rough, one of the alternative goals for a recursive call may be to leave the result in a temporary stack location, even though based on the previous analysis, this should not be necessary. (This is true for the PDP11 at least. See for example the machine-dependent routines `setbin` (3525) and `nextcook` (4075), where the alternative cookie can include `INTEMP` (0387).)

Clearly the dividing line between what will, and can, be matched via templates, and what will be handled via special cases recognized by the program, is not obvious.

6.3 Code Sections

The code for `order` divides into four main sections:

1534	again:	INITIALIZATION
1549		
1555	switch	For most operators, call `match`. (several times, if it seems appropriate).
1586		
1587		The hard slog. Lots of special cases
1601	switch	for rewriting the tree which end variously with either a transfer to one of `again`, `nomat`, or `cleanup`,
1778		or a `return`, or a call to `cerror`.
1780	cleanup:	
1799		THE USUAL WAY OUT

1526: One may wonder about the choice of the variables which have been assigned as register variables in this procedure ... ?

6.4 First Section

1536: Copy `cook` to `cookie`. (It may be overwritten at line 1563.)

1537: `rcount` (1516) performs a safety check.

1538: `canon` (1307) will recheck to see if new OREG nodes can be created, and will recalculate the SU numbers.

1539: `rallo` (3006) is a machine dependent routine that performs register allocation. Prior to this point the SU numbers have provided estimates of the numbers of registers required, without the verification that a feasible allocation (giving regard to the idiosyncrasies of the actual cpu) will be possible. Its job is to set the `rall` field for those nodes of the subtree (whose root is passed as a parameter) to ensure that the final value calculated will appear in the appropriate register, without over-constraining the association of registers with the remaining nodes.

6.5 Second Section

1555: The `switch` which begins here effectively preempts the `table` search for the set of operators that are listed beginning at line 1571.

1557: Except for the 13 operator types listed on lines 1571 to 1583, call `match` (2159) hopefully to generate code for the subtree. It will return one of several results:

MDONE	plain sailing
MNOPE	Doesn't look promising. Moderate your expectations as to where the calculated result may be stored.
other	Reorganize the calculation (i.e. re-write the tree), via one of the special cases handled in the next section.

1559: Structured programming enthusiasts would most probably prefer to see the next ten lines replaced by something like:

```
for( ;; ){
        m = match( p, cookie );
        if( m == MDONE ) goto cleanup;
        if( m != MNOPE ) break;
        cookie = nextcook( p, cookie );
        if( !cookie ) goto nomat;
        }
```

1571: The first seven operators in the list that starts here are not to be matched explicitly. FORCE is a pseudo-operator which exists to signal that certain values must appear in pre-defined places (in particular the result returned by a procedure must appear in register R0 or FR0).

1578: The remaining six operators can be re-interpreted in a machine independent fashion into more basic operations. The procedure call operators are not to be matched explicitly at this point. Instead the match will be made via a call to `match` from `gencall` (4032).

6.6 Third Section

1590: Begin by setting p1 and p2, and doing a little diagnostic printing, if appropriate.

1606: This code is for COMOPs which escaped the net cast by `delay1` because they occurred in the right subtree of a QUEST or ANDAND or OROR operator.

```
	if( odebug ){
		printf( "order( %o, ", p );
		prcook( cook );
		printf( " ), cookie " );
		prcook( cookie );
		printf( ", rewrite %s\n", opst[m] );
		}
	switch( m ){
	default:
	nomat:
		cerror( "no table entry for op %s", opst[p->op] );

	case COMOP:
		codgen( p1, FOREFF );
		p2->rall = p->rall;
		codgen( p2, cookie );
		ncopy( p, p2 );
		p2->op = FREE;
		goto cleanup;

	case FORCE:
		/* recurse, letting the work be done by rallo */
		p = p->left;
		cook = INTAREG|INTBREG;
		goto again;

	case CBRANCH:
		o = p2->lval;
		cbranch( p1, -1, o );
		p2->op = FREE;
		p->op = FREE;
		return;

	case QUEST:
		cbranch( p1, -1, m=getlab() );
		p2->left->rall = p->rall;
		codgen( p2->left, INTAREG|INTBREG );
		/* force right to compute result into same register
			as used by left */
		p2->right->rall = p2->left->rval|MUSTDO;
		reclaim( p2->left, RNULL, 0 );
		cbgen( 0, m1 = getlab(), 'I' );
		deflab( m );
		codgen( p2->right, INTAREG|INTBREG );
		deflab( m1 );
		p->op = REG;  /* set up node describing result */
		p->lval = 0;
		p->rval = p2->right->rval;
		p->type = p2->right->type;
		tfree( p2->right );
		p2->op = FREE;
		goto cleanup;

	case ANDAND:
	case OROR:
	case NOT:  /* logical operators */
		/* if here, must be a logical operator for 0-1 value */
		cbranch( p, -1, m=getlab() );
		p->op = CCODES;
		p->label = m;
		order( p, INTAREG );
		goto cleanup;

	case FLD:   /* fields of funny type */
		if ( p1->op == UNARY MUL ){
			offstar( p1->left );
			goto again;
			}

	case UNARY MINUS:
		order( p1, INBREG|INAREG );
		goto again;
```

1607: Call codgen to handle the left subtree, for effect only (i.e. any result calculated does not need to be saved).

1608: Transfer information regarding the preferred location of the result from the root to the root of the right subtree.

1609: Call codgen to handle the right subtree. The value obtained will be the value of the whole expression.

1610: Copy information about the value calculated by the right subtree into the root node.

codgen, not order, is called at lines 1607 and 1609 to handle the subtree, because the call to store in the earlier invocation of codgen (dormant and not yet complete) will not have explored properly below the COMOP. A similar comment also applies to lines 1630 and 1637.

1615: rallo (3006) will recognize FORCE as a special case, and will mark the root node of the left subtree (UTYPE operator) as either R0|MUSTDO or FR0|MUSTDO, whichever is appropriate.

1620: cbranch (1806) generates code to evaluate the subtree designated by its first argument, and generates a branch instruction which will be taken at object time if the result is true, and another to be taken if the result is false. If either of the label numbers supplied as the second and third arguments is negative, no explicit branch statement will be needed, and in the appropriate circumstances, control will "fall through" to the next statement.

6.7 Conditional Operators

1627: QUEST is the operator for conditional expressions. The code on lines 1628 to 1644 is a textbook case of the use of the piece parts provided in the compiler.

1628: cbranch (1806) is being asked to take the left subtree, evaluate it, and if the result is true, fall through, otherwise generate a forward branch to the label returned by getlab (3353). (This label will be placed in the assembler output at line 1636.)

1629: Copy the rall value into the root of the left sub-subtree of the right subtree. This implies a stronger condition than rallo would have applied to this node. (Note that the root node of the right subtree should be a COLON.)

1630: Generate code for the true case.

1633: Whatever the location of the result from the left subtree, make the right subtree put its result in the same place.

1634: reclaim (2677), in this situation, will "unmark" any registers used from the evaluation of the left subtree of p2, and will free all the nodes in that subtree.

1635: Generate an unconditional branch around the code for the false case.

1636: Place the label generated at line 1628 into the assembler output to mark the beginning of the code for the false case.

1637: Generate code for the false case.

1638: Place the label after the code for the false case.

1639: Replace the subtree designated by `p` by single leaf node, which represents the register into which the result was forced by the action at line 1633.

1651: The relevant code in `cbranch` for this case begins at line 1872. The result is available in the condition codes, and the subtree is reduced to a single node.

1654: The recursive call on `order` will result in a template match (see lines 4970 through 4980).

6.8 Some Miscellaneous Cases

1657: FLD and the next three case are not mentioned in the original list (lines 1571-1583) of special cases. They can only occur as values returned by `match`. Get the left hand operand into a directly addressable form if it is not already so. `offstar` (3363) will attempt to get the left subtree into the form of an OREG. Note that if the tree does not begin with a UNARY MUL, fall through to the next case.

1663: If we can't get the negative value where it is needed directly, calculate its complement, by a recursive call to `order` (the SU numbers etc. should still be ok), and then reverse the sign via another iteration of the present `order` invocation. The next time there should be a matching template.

1667: Things (i.e. subtrees) that should be placed in a register may get here. However if the node is already a REG, something has gone wrong ...

1673: `INIT` operators should always have a left subtree which is a constant, and should be matched by a template. Any other situation is a compiler error (not a user error). See the templates on lines 5444 through 5454.

6.9 Procedure Calls

1677: Since, for the PDP11 and VAX11/780, `genfcall` and `genscall` are defined to be `gencall` (4032), the code from here to line 1696 is over elaborate for these machines.

1681: `genfcall` is defined to be identical to `gencall`. Also `genscall` on line 1695 reduces to `gencall`, so the differentiation amongst the types of call, at least at this point, is unnecessary for the PDP11 and VAX11/780.

1703: A small optimization.

1709: `offstar` (3363) attempts to transform the subtree under a UNARY MUL into a form that will be reduced to an OREG by `canon` (1307).

1713: For the PDP11 and VAX11/780, `setincr` is a no-op and always returns the value zero, i.e. there is no special processing. (There is most probably an opportunity to refine the compiler at this point. Contrast this case with the next case, for STASG on line 1731.)

`setincr` (3378) is the first of a set of procedures with names beginning with `set`. These are called to provide machine-dependent recognition and processing of various cases, before the more general machine-independent processing occurs.

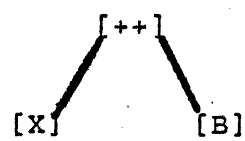
One does not gain a clear understanding of the function of e.g. `setincr` by studying the version of it needed for the PDP11. However its function is to apply ad hoc rewriting rules to the

```
	case ASG PLUS:  /* and other assignment ops */
		if( setasop(p) ) goto again;

		/* there are assumed to be no side effects in LHS */

		p2 = tcopy(p);
		p->op = ASSIGN;
		reclaim( p->right, RNULL, 0 );
		p->right = p2;
		canon(p);
		rallo( p, p->rall );

		if( odebug ) fwalk( p, eprint, 0 );

		order( p2->left, INTBREG|INTAREG );
		order( p2, INTBREG|INTAREG );
		goto again;

	case ASSIGN:
		if( setasg( p ) ) goto again;
		goto nomat;


	case BITYPE:
		if( setbin( p ) ) goto again;
		/* try to replace binary ops by =ops */
		switch(o){

		case PLUS:
		case MINUS:
		case MUL:
		case DIV:
		case MOD:
		case AND:
		case OR:
		case ER:
		case LS:
		case RS:
			p->op = ASG o;
			goto again;
			}
		goto nomat;

		}

	cleanup:

	/* if it is not yet in the right state, put it there */

	if( cook & FOREFF ){
		reclaim( p, RNULL, 0 );
		return;
		}

	if( p->op==FREE ) return;

	if( tshape( p, cook ) ) return;

	if( (m=match(p,cook) ) == MDONE ) return;

	/* we are in bad shape, try one last chance */
	if( lastchance( p, cook ) ) goto again;

	goto nomat;
	}

/* ============================================ */

```

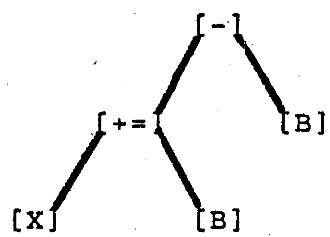
current tree, at or near its root, which will take advantage of, or disguise the deficiencies of, a particular target machine. (Just how to recognize such cases in the first place is another problem!)

1717: Convert the operation to ASG PLUS or ASG MINUS. If the value is not needed further, just change the operator type of p; else ...

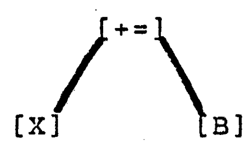
1724: Rewrite the subtree with additional nodes so that



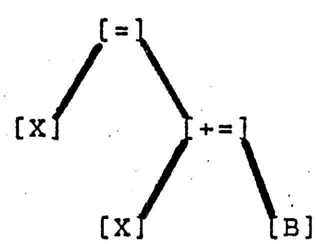
becomes



1740: Rewrite the subtree with additional nodes, so that



becomes



1743: `p2->op` is not reset at this point because, after the call to `order` at line 1749, the ASG PLUS operator will be applied to a copy of the value of the left subtree that resides in a temporary register.

1758: `BITYPE` is a value which can be returned by `match` (2159) as a result of the "last-ditch" template at line 5515. `setbin` (3525) attempts to rewrite the tree in successive stages until either a call to `match` results in a successful template match.

Fortunately, the remainder of this file is not so heavy going as the earlier parts. The three remaining procedures are quite distinct, and pleasantly different:

1. `cbranch` (1806) generates code for conditional branches.
2. `ffld` (1928) rewrites the tree to handle field extractions when there is no hardware to do the job.
3. `oreg2` (1988) rewrites the tree so that address arithmetic will be done implicitly by the hardware, whenever possible.

7.1 `negrel` (1804)

This array is used for reversing the sense of relational tests. Its contents can be understood from the definitions on lines 0092 to 0101 for the ten relational operator types, viz.

EQ	80	GT	85
NE	81	ULE	86
LE	82	ULT	87
LT	83	UGE	88
GE	84	UGT	89

The reverse of `EQ` is `NE == negrel [EQ - EQ]`, of `ULT`, `UGE == negrel [ULT - EQ]`, etc.

7.2 `cbranch` (1806)

This procedure is called by `order` from three different locations. It generates a conditional branch instruction which will use the result to be calculated from the tree which is passed as the first parameter. In all three cases, the second parameter is `-1`, which implies that no branch is to be taken if the result is true. `cbranch` also calls itself recursively at several places. It uses the machine-dependent routine `cbgen` (3981) to emit the actual assembler branch instruction.

1807: See comment in the code.

1826: This code is used to standardize the situation, when one of the alternatives is to "fall through" to the next instruction. Arrange things, by reversing the sense of the test if necessary, so that the "fall through" path will always be the false path.

1831: `NOOPT` is not set for the PDP11, so keep going to give special treatment for comparisons against zero.

1832: If the right operator is a genuine constant zero ...

1833: confirmed by a null name! ...

1834: rewrite the operation.

```
	/* negatives of relationals */
int negrel[] = { NE, EQ, GT, GE, LT, LE, UGT, UGE, ULT, ULE } ;

cbranch( p, true, false ) NODE *p; {
	/* evaluate p for truth value, and branch to true or false
	/* accordingly: label <0 means fall through */

	register o, lab, flab, tlab;

	lab = -1;

	switch( o=p->op ){

	case ULE:
	case ULT:
	case UGE:
	case UGT:
	case EQ:
	case NE:
	case LE:
	case LT:
	case GE:
	case GT:
		if( true < 0 ){
			o = p->op = negrel[ o-EQ ];
			true = false;
			false = -1;
			}
# ifndef NOOPT		>>>>>>>>>>>>>>>>>>>>>
		if( p->right->op == ICON && p->right->lval == 0 &&
				p->right->name[0] == '\0' ){
			switch( o ){

			case UGT:
			case ULE:
				o = p->op = (o==UGT)?NE:EQ;
			case EQ:
			case NE:
			case LE:
			case LT:
			case GE:
			case GT:
				if( logop(p->left->op) ){
			/* strange situation: e.g., (a!=0) == 0; must
			/* prevent reference to p->left->label, so get 0/1
			/* we could optimize, but why bother */
					codgen( p->left, INAREG|INBREG );
					}
				codgen( p->left, FORCC );
				cbgen( o, true, 'I' );
				break;

			case UGE:
				/* unconditional branch */
				cbgen( 0, true, 'I' );
			case ULT:
				;   /* do nothing for LT */
				}
			}
		else
# endif			<<<<<<<<<<<<<<<<<<<<<<
			{
			p->label = true;
			codgen( p, FORCC );
			}
		if( false>=0 ) cbgen( 0, false, 'I' );
		reclaim( p, RNULL, 0 );
		return;

```

1838: Unsigned comparisons against zero can be converted to signed comparisons against zero: `UGT` becomes `NE`, `ULE` becomes `EQ`.

1851: Generate code to evaluate the left subtree into the condition bits of the processor status word.

1851: `codgen` will cause the root node operator to be changed to `CCODES`.

1852: Call `cbgen` (3981), passing the operation, `o`, as the first parameter. `o` is the original root node operator. The parameter `'I'` implies the regular case for `cbgen`.

1855: A `UGE` comparison against zero must always succeed, so generate an unconditional branch. (Note that the initial argument to `cbgen` is zero.)

1858: A `ULT` comparison against zero must always fail. So do nothing, and "fall through".

1864: This is the normal case (also the unoptimized test against zero). Copy the "true" label into the `label` field of the root node, and call `codgen` with the cookie `FORCC` i.e. the goal of leaving the result in the condition codes. Note that the value of `label` will be picked up by `zzzcode` at line 4426.

1868: Generate the false branch, if needed.

1869: Call `reclaim` with the argument `RNULL` to completely dismantle the subtree.

1872: The conjunction of two conditions is false if the first condition is false. Transform the tree so that the equivalent of

```
        if (A && B) {goto true;} else {goto false;}
```

will become the equivalent of

```
        if (! A) {goto false;}
        if (B) {goto true;} else {goto false;}
```

1873: If false is intended to refer to the next statement (i.e. the label for the false branch is negative) use `getlab` to provide a unique new label number.

1874: Call `cbranch` recursively twice in succession to generate code for the two equivalent branch statements shown above.

1876: If a label was generated, call `deflab` (3358) to declare it at the current location in the assembler output file.

1880: This case is handled analogously to the `ANDAND` case.

1888: Call `cbranch` recursively with the left subtree as the first argument, and with the other arguments reversed. A textbook application of recursion.

1893: This case is also a textbook variety. There seems to be nothing significant about freeing the root node before, rather than after, the recursive call on `cbranch`.

1899: This case is handled also by rewriting the tree in a way analogous to rewriting

```
        if (p ? l : r) {goto true;} else {goto false;}
```

into the form

```
        if (not p) {goto z;}
        if (l) {goto true;} else {goto false;}
 z:
        if (r) {goto true;} else {goto false;}
```

```
	case ANDAND:
		lab = false<0 ? getlab() : false ;
		cbranch( p->left, -1, lab );
		cbranch( p->right, true, false );
		if( false < 0 ) deflab( lab );
		p->op = FREE;
		return;

	case OROR:
		lab = true<0 ? getlab() : true;
		cbranch( p->left, lab, -1 );
		cbranch( p->right, true, false );
		if( true < 0 ) deflab( lab );
		p->op = FREE;
		return;

	case NOT:
		cbranch( p->left, false, true );
		p->op = FREE;
		break;

	case COMOP:
		codgen( p->left, FOREFF );
		p->op = FREE;
		cbranch( p->right, true, false );
		return;

	case QUEST:
		flab = false<0 ? getlab() : false;
		tlab = true<0 ? getlab() : true;
		cbranch( p->left, -1, lab = getlab() );
		cbranch( p->right->left, tlab, flab );
		deflab( lab );
		cbranch( p->right->right, true, false );
		if( true < 0 ) deflab( tlab);
		if( false < 0 ) deflab( flab );
		p->right->op = FREE;
		p->op = FREE;
		return;

	default:
		/* get condition codes */
		codgen( p, FORCC );
		if( true >= 0 ) cbgen( NE, true, 'I' );
		if( false >= 0) cbgen(true >= 0 ? 0 : EQ, false, 'I');
		reclaim( p, RNULL, 0 );
		return;

		}

	}
/* ---------------------------------------------------- */

# ifndef FIELDOPS       >>>>>>>>>>>>>>>>>>>>>>
	/* do this if no special hardware support for fields */

ffld( p, down, down1, down2 ) NODE *p; int *down1, *down2; {
	 /* look for fields that are not in an lvalue context,
		and rewrite them... */
	register NODE *shp;
	register s, o, v, ty;

	*down1 =  asgop( p->op );
	*down2 = 0;

	if( !down && p->op == FLD ){ /* rewrite the node */

		if( !rewfld(p) ) return;

		ty = (szty(p->type) == 2)? LONG: INT;
```

1912: In the remaining cases, the value to be tested is not a logical expression. Evaluate the subtree so as to set the condition codes.

1915: Generate the true branch if required.

1916: Generate the false branch if and as appropriate.

7.3 `ffld` (1928)

This procedure is used when there is no special hardware for extracting subfields of memory words, and the desired effect must be obtained by masking and shifting. This is true for the PDP11 but not the VAX11/780. `ffld` is invoked at each node via `fwalk` (0699), from a call from `canon` at line 1313.

1934: If the operator for the current node is an assignment operator, pass this value on to the procedure invocation that will process the left subtree, when its time comes.

1935: The right subtree requires no special treatment.

1937: If this is not the left subtree of an assignment operator, and the current operator is a `FLD`, then there is work to be done.

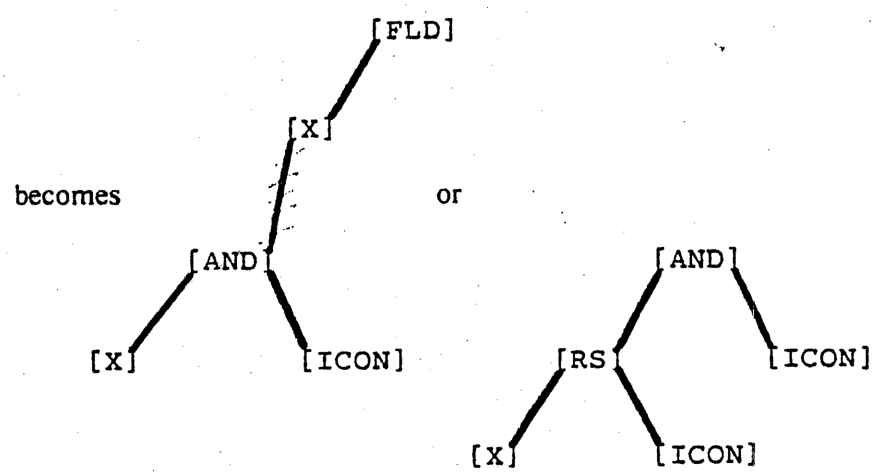
1939: `rewfld` is a machine-dependent procedure (which is a no-op for the PDP11) which gets a chance at this point to apply any special tricks which work in limited situations, e.g. if hardware exists to extract a character from a word. (Such hardware is available on the Honeywell 6000, for example.)

1941: Treat data as an integer or long*, i.e. forget refinements such as floating point or unsigned integers.

1942: `v` contains two fields: the least significant field of six bits defines the size of target field, and it is unpacked via the macro `UPKFSZ`; the other field, by `UPKFOFF` (0232).

1945: `o` defines the offset of the field within the word.

1952: Rewrite the tree so that



* Fields for long variables are not implemented on the PDP11, though they conceivably could be (hlb).

```
                v = p->rval;
                s = UPKFSZ(v);
# ifdef RTOLBYTES      >>>>>>>>>>>>>>>>>>>>>>>
                o = UPKFOFF(v);  /* amount to shift */
# else                 xxxxxxxxxxxxxxxxxxxxxxxx
                o = szty(p->type)*SZINT - s - UPKFOFF(v);
# endif                <<<<<<<<<<<<<<<<<<<<<<

                /* make & mask part */

                p->left->type = ty;

                p->op = AND;
                p->right = talloc();
                p->right->op = ICON;
                p->right->rall = NOPREF;
                p->right->type = ty;
                p->right->lval = 1;
                p->right->rval = 0;
                p->right->name[0] = '\0';
                p->right->lval <<= s;
                p->right->lval--;

                /* now, if a shift is needed, do it */

                if( o != 0 ){
                        shp = talloc();
                        shp->op = RS;
                        shp->rall = NOPREF;
                        shp->type = ty;
                        shp->left = p->left;
                        shp->right = talloc();
                        shp->right->op = ICON;
                        shp->right->rall = NOPREF;
                        shp->right->type = ty;
                        shp->right->rval = 0;
                        shp->right->lval = o;  /* amount to shift */
                        shp->right->name[0] = '\0';
                        p->left = shp;
                        /* whew! */
                        }
                }
        }
# endif                <<<<<<<<<<<<<<<<<<<<<<
/* ------------------------------------------------------- */

oreg2( p ) register NODE *p; {

        /* look for situations where we can turn * into OREG */

        NODE *q;
        register i;
        register r;
        register char *cp;
        register NODE *ql, *qr;
        CONSZ temp;

        if( p->op == UNARY MUL ){
                q = p->left;
                if( q->op == REG ){
                        temp = q->lval;
                        r = q->rval;
                        cp = q->name;
                        goto ormake;
                        }

                if( q->op != PLUS && q->op != MINUS ) return;
                ql = q->left;
                qr = q->right;

```

1952: Set the type field for the left descendent.

1954: Change the root node operator from a unary `FLD` to a binary `AND`.

1955: Create a new node to hold the mask.

1963: Finally (!), the value is `(1<<s) - 1`

1967: If a shift is needed, introduce an extra node (`RS`) with its associated right descendent, of type `ICON` (a constant to specify the number of shift positions), so that the subtree becomes the second case shown above.

Fields which occur as part of an "lvalue" are handled directly in the code templates. Note also that the rewriting of an `INCR` (`++`) or `DECR` (`--`) operator can cause the appearance of field operators on the right-hand side of a subtree. Hence the scanning for field operators is carried out repeatedly, whenever `canon` (1307) is called, during the reduction of the expression tree.

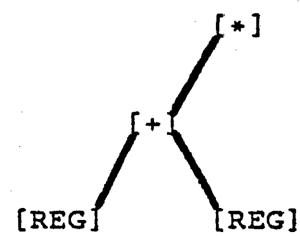
7.4 `oreg2` (1988)

This procedure is called indirectly from `canon` (1307) via a call to `walkf` (0688). The routine traverses the expression tree in endorder, looking for ways to eliminate explicit additions or subtractions which occur during address calculations in the hope of delegating these to the hardware addressing mechanisms.

There are four types of subtree which may be sought out and transformed into a single node of type `OREG`. With `PM` used to denote either `PLUS` or `MINUS`, these are as follows:

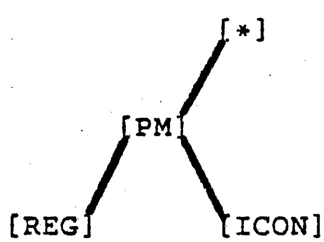
(i)

(ii)

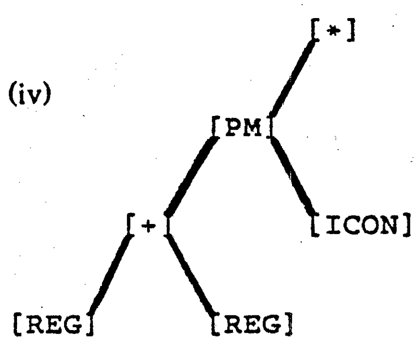


and

(iii)



(iv)



2001: Case (i)

2005: The label `ormake` occurs at line 2064.

2012: Machines with the hardware for double indexing include the IBM 360/370 and the Interdata 8/32, but not the PDP11 or VAX11/780.

```
# ifdef R2REGS          >>>>>>>>>>>>>>>>>>>>>>>>

                /* look for doubly indexed expressions */

                if( q->op==PLUS && qr->op==REG && ql->op==REG &&
                                (szty(ql->type)==1||szty(qr->type)==1) ) {
                        temp = 0;
                        cp = ql->name;
                        if( *cp ){
                                if( *qr->name ) return;
                                }
                        else {
                                cp = qr->name;
                                }
                        if( szty(qr->type)>1)
                                r = R2PACK(qr->rval,ql->rval);
                        else r = R2PACK(ql->rval,qr->rval);
                        goto ormake;
                        }

                if( (q->op==PLUS||q->op==MINUS) && qr->op==ICON &&
                                ql->op==PLUS && ql->left->op==REG &&
                                ql->right->op==REG ){
                        temp = qr->lval;
                        cp = qr->name;
                        if( q->op == MINUS ){
                                if( *cp ) return;
                                temp = -temp;
                                }
                        if( *cp ){
                                if( *ql->name ) return;
                                }
                        else {
                                cp = ql->name;
                                }
                        r = R2PACK(ql->left->rval,ql->right->rval);
                        goto ormake;
                        }

# endif                 <<<<<<<<<<<<<<<<<<<<<<

                if( (q->op==PLUS || q->op==MINUS) && qr->op == ICON &&
                                ql->op==REG && szty(qr->type)==1) {
                        temp = qr->lval;
                        if( q->op == MINUS ) temp = -temp;
                        r = ql->rval;
                        temp += ql->lval;
                        cp = qr->name;
                        if( *cp && ( q->op == MINUS || *ql->name ) )
                                return;
                        if( !*cp ) cp = ql->name;

        ormake:
                        if( notoff( p->type, r, temp, cp ) ) return;
                        p->op = OREG;
                        p->rval = r;
                        p->lval = temp;
                        for( i=0; i<NCHNAM; ++i )
                                p->name[i] = *cp++;
                        tfree(q);
                        return;
                        }
                }
        }
/* ------------------------------------------------- */
```

2018: Case (ii). Not supported by the PDP11 or VAX11/780.

2035: Case (iv). Not supported by the PDP11 or VAX11/780.

2055: Case (iii) is fairly straightforward.

2060: If `*cp` is non-null, then the constant is an array address. Do not subtract it from anything, or add it to another array name.

2065: Check the size of the offset and abandon the attempt if is out of range. (Not a problem on the PDP11 or VAX11/780, but a real one on the IBM 360/370 where the offset must satisfy $0 <= k < 4096$.)

2066: Respecify the root node.

2071: Throw away the former left subtree.

```
# include "mfile2"

int fldsz, fldshf;

		/* masks for matching dope with shapes */
static int mamask[] = {
		SIMPFLG,		/* OPSIMP */
		SIMPFLG|ASGFLG,	/* ASG OPSIMP */
		COMMFLG,		/* OPCOMM */
		COMMFLG|ASGFLG,	/* ASG OPCOMM */
		MULFLG,		/* OPMUL */
		MULFLG|ASGFLG,	/* ASG OPMUL */
		DIVFLG,		/* OPDIV */
		DIVFLG|ASGFLG,	/* ASG OPDIV */
		UTYPE,		/* OPUNARY */
		TYFLG,		/* ASG OPUNARY is senseless */
		LTYPE,		/* OPLEAF */
		TYFLG,		/* ASG OPLEAF is senseless */
		0,		/* OPANY */
		ASGOPFLG|ASGFLG,/* ASG OPANY */
		LOGFLG,		/* OPLOG */
		TYFLG,		/* ASG OPLOG is senseless */
		FLOFLG,		/* OPFLOAT */
		FLOFLG|ASGFLG,	/* ASG OPFLOAT */
		SHFFLG,		/* OPSHFT */
		SHFFLG|ASGFLG,	/* ASG OPSHIFT */
		SPFLG,		/* OPLTYPE */
		TYFLG,		/* ASG OPLTYPE is senseless */
		};
/* -------------------------------------------------- */

struct optab *rwtable;

struct optab *opptr[DSIZE];

setrew(){
	/* set rwtable to first value which allows rewrite */
	register struct optab *q;
	register int i;

	for( q = table; q->op != FREE; ++q ){
	    if( q->needs == REWRITE ){
		rwtable = q;
		goto more;
		}
	    }
	cerror( "bad setrew" );


	more:
	for( i=0; i<DSIZE; ++i ){
	    if( dope[i] ){ /* there is an op... */
		for( q=table; q->op != FREE; ++q ){
		    /*	beware; things like LTYPE that match
			multiple things in the tree must
			not try to look at the NIL at this
			stage of things!  Put something else
			first in table.c  */
		/* at one point, the operator matching was 15%
		    of the total compile time; thus, the function
		    call that was here was removed...
		    */

		    if( q->op < OPSIMP ){
			if( q->op==i ) break;
			}
```

```
                        else {
                            register opmtemp;
                            if((opmtemp=mamask[q->op - OPSIMP])&SPFLG){
                                if( i==NAME || i==ICON || i==OREG ) break;
                                else if( shltype( i, NIL ) ) break;
                                }
                            else if( (dope[i]&(opmtemp|ASGFLG)) ==
                                                opmtemp ) break;
                            }
                        }
                    opptr[i] = q;
                    }
                }
            }
/* ================================================== */

match( p, cookie ) NODE *p; {
        /* called by: order, gencall
           look for match in table and generate code if found,
           unless entry specified REWRITE.  Returns MDONE, MNOPE,
           or rewrite specification from table */

        register struct optab *q;
        register NODE *r;

        rcount();
        if( cookie == FORREW ) q = rwtable;
        else q = opptr[p->op];

        for( ; q->op != FREE; ++q ){

            /* at one point the call that was here was over 15% of
                the total time;
                thus the function call was expanded inline */
            if( q->op < OPSIMP ){
                if( q->op!=p->op ) continue;
                }
            else {
                register opmtemp;
                if((opmtemp=mamask[q->op - OPSIMP])&SPFLG){
                    if( p->op!=NAME && p->op!=ICON && p->op!= OREG &&
                        ! shltype( p->op, p ) ) continue;
                    }
                else if( (dope[p->op]&(opmtemp|ASGFLG)) != opmtemp )
                            continue;
                }

            if( !(q->visit & cookie ) ) continue;
            r = getlr( p, 'L' );  /* see if left child matches */
            if( !tshape( r, q->lshape ) ) continue;
            if( !ttype( r->type, q->ltype ) ) continue;
            r = getlr( p, 'R' );  /* see if right child matches */
            if( !tshape( r, q->rshape ) ) continue;
            if( !ttype( r->type, q->rtype ) ) continue;

                /* REWRITE means no code from this match but go
                   ahead, and rewrite node to help future match */
            if( q->needs & REWRITE ) return( q->rewrite );
                /* if can't generate code, skip entry */
            if( !allo( p, q ) ) continue;

            /* resources are available; generate code */
            expand( p, cookie, q->cstring );
            reclaim( p, q->rewrite, cookie );
            return(MDONE);
            }
        return(MNOPE);
        }
/* -------------------------------------------------- */
```

This file contains procedures that are concerned with matching the templates in the `table` array with the operations required by the expression tree. A close perusal of this file will persuade the reader that there are a number of implicit assumptions made in the code regarding the contents of `table`. Moreover, since a good part of the code is machine dependent, the set of implicit assumptions is also machine dependent. (Not an altogether desirable situation, and a hard act to follow!)

The procedures in the file are as follows:

1. `setrew` finds appropriate starting points for searching `table`.
2. `match` searches `table` looking for a template which matches in all relevant respects.
3. `getlr` returns a pointer to a node that is related to the current node (usually a child).
4. `tshape` compares the shape of a tree with a set of possible shapes.
5. `ttype` compares operand types with operation capabilities.
6. `expand` expands a character string into a set of assembler instructions.

The initial entry in this file is the declaration of two integer variables, `fldsz` and `fldshf`, which are used in connection with bit fields. They are set as a side-effect of `tshape`, and used by `expand`.

The next entry is the declaration and initialization of the array `mamask` (2082). This provides, for each of several groups of related operators, a bit mask compatible with the ones stored in `dope` (0724) for simple operators. The comments on lines 2083 through 2104 can be better understood if reference is made to the definitions of `OPSIMP`, `OPCOMM`, ... `OPLTYPE` on lines 0360 to 0370. Note that these are even integers and that `ASG` is defined as "`1 +`".

8.1 `setrew` (2112)

This procedure, which is called once by `p2init` at line 0956, searches the array `table` (4669), which contains all the operator templates. (The initialization of `table` occupies a file of its own, `table.c`.)

2117: The first task is to locate an entry for which the `needs` field has the value `REWRITE`, and to store a pointer to this entry.

2123: If no such entry can be found, this is taken to imply a fatal defect in the file `table.c`, and hence a compiler error, since the perfect computer, with an operator for every occasion, has not yet been invented.

2126: Then for each operator in turn ...

2128: which is valid (i.e. has an entry in the array `dopest` (0727), and hence a non-zero entry in `dope` (0724)) ...

2129: look through the entries in `table` and ...

2140: if the `table` entry is for a simple operator, and matches the current operator ...

2153: store a pointer to the `table` entry in the `opptr` (2110) array. This will provide a starting point for search of `table` when the particular operator is to be matched.

2144: If the operator type found in the `table` entry refers to a group of operators, i.e. has a value in the range `OPSIMP` through `ASG OPLTYPE`, then subtract `OPSIMP` from it, and using the result as an index into `mamask`, retrieve the corresponding bit mask.

2146: If the `SPFLG` is set (true only for `OPLTYPE` i.e. a leaf node) then if the node is a `NAME` or a constant or an `OREG`, then a starting point has been found that can be recorded in `opptr`.

2147: The `SPFLG` is set but the previous test did not succeed. Call the machine dependent routine `shltype` (4141) to make a determination as to whether the operator has the shape of a leaf. (An examination of `shltype` for this version of the compiler shows repetition of the code of line 2146, and only one extra case, `REG`, being recognized.)

2149: If the `SPFLG` was not set, see if bits set in `mamask` entry are matched by corresponding bits in the `dope` (0724) entry for the operator, with the added proviso that, if the operator to be matched is an assignment operator, the operator group must also represent assignment operators.

The last entry in `table` is an entry for the operator `FREE`, and the search of the table stops there. Hence, if for a particular operator, there is no `table` entry that matches the operator, the corresponding entry in `opptr` will have the value `FREE`.

8.2 `match` (2159)

This procedure is called by `order` (also `gencall` (4032)) to try and find a template in the `table` which matches the operation defined by the subtree whose root is passed as the first argument.

2168: Check to see if there have been too many iterations.

2169: Determine the starting point for the search from the values prepared by `setrew`.

2172: The `for` loop which begins here bears more than a passing resemblance to the loop in `setrew`. However instead of breaking from the loop when the first matching entry is found, the requirement now is to ignore (`continue`) entries which do not match.

2178: Matching the template requires satisfactory answers to a series of questions. The first question is whether the operation defined in the template is, or includes, the operation to be matched.

2190: The next question is whether the template defines an operation which can create the desired effect (meet the required goal).

2191: Look at the left descendent, and see if ...

2192: the "shape" of the left subtree is compatible with the "shape" of the left operand of the table entry. If so ...

```
NODE *
getlr( p, c ) NODE *p; {

        /* return the pointer to the left or right side of p,
          or p itself, depending on the optype of p */

        switch( c ) {

        case '1':
        case '2':
        case '3':
            return( &resc[c-'1'] );

        case 'L':
            return( optype( p->op ) == LTYPE ? p : p->left );

        case 'R':
            return( optype( p->op ) != BITYPE ? p : p->right );

            }
        cerror( "bad getlr: %c", c );
        /* NOTREACHED */
        }
/* -------------------------------------------------- */

tshape( p, shape ) NODE *p; {
        /* return true if shape is appropriate for the node p
           side effect for SFLD is to set up fldsz,etc */
        register o, mask;

        o = p->op;
        if( sdebug ){
            printf( "tshape( %o, %o), op = %d\n", p, shape, o );
            }

        if( shape & SPECIAL ){

            switch( shape ){

            case SZERO:
            case SONE:
            case SMONE:
                if( o != ICON || p->name[0] ) return(0);
                if( p->lval == 0 && shape == SZERO ) return(1);
                else if( p->lval == 1 && shape == SONE ) return(1);
                else if( p->lval == -1 && shape == SMONE ) return(1);
                else return(0);

            default:
                return( special( p, shape ) );
                }
            }

        if( shape & SANY ) return(1);

        if( (shape&INTEMP) && shtemp(p) ) return(1);

        if( (shape&SWADD) && (o==NAME||o==OREG) ){
            if( BYTEOFF(p->lval) ) return(0);
            }

        switch( o ){

        case NAME:
            return( shape&SNAME );
        case ICON:
            mask = SCON;
            return( shape & mask );

```

2193: are the types compatible?

The last few lines raise a number of points for discussion. First, the term "shape" in this context refers to the set of locations in which an operand can occur, e.g. in a register, or in the temporary part of the stack, etc., or some combination of these. This is to be distinguished from "type", which refers to the category of information stored, its representation, and its size.

Second, the use of the procedure `getlr` (2214) seems curious. `getlr` will return a reference to the left subtree, if one exists, otherwise a reference to the *node itself.* If the current node is `BITYPE`, then, clearly, tests are going to be applied to both the left and right subtrees of the node. If the current node is `UTYPE`, then tests for shape and operand type are going to be applied to the left subtree and the node itself. If the `UTYPE` node is a type conversion, for instance, then it is quite useful to be able to do this. Finally, if the node is `LTYPE`, tests are going to be applied against the node itself twice. While one set of these may be useful, the second set is certainly going to be redundant. (See, for example, the group of templates starting at line 4897.)

2194: Perform shape and type compatibility tests on the right hand side.

2200: Certain general entries are used to recognize situations where the tree should be reorganized, e.g. by introducing additional nodes which represent actions which can be avoided on some machines.

2202: Call `allo` to allocate resources, i.e. temporary registers and/or temporary space in the object stack.

2205: With all barriers surmounted at last, call `expand` (2376) to take the string which is the last item in the `table` entry, and expand it macro-fashion into one or more lines of assembler code.

2206: Call `reclaim` (2677) to rewrite the tree to reflect the progress made in the calculation by the code just emitted, and to return any resources no longer reserved back to the free lists.

2207: Report success.

2209: The whole table has been searched without success. Report failure.

8.3 `getlr` (2214)

The functioning of this procedure is clear enough. It attempts to return a valid node pointer in all situations. The use of a character value as the second argument, rather than a binary value, is dictated by the needs of `expand`, e.g. at line 2428, and its alter ego, `zzzcode`, which extract the argument from a character string.

8.4 `tshape` (2238)

This procedure is called with two arguments: a node pointer, and a "shape", which is a statement about the possible forms that the data that the node represents can assume. Since the matching of templates is not done recursively, the node had better represent something directly accessible, not something which can be computed in principle. As has been seen, `tshape` is called by `order` (line 1791) with a second argument `cook` to see if the goal can be satisfied by the current node without ever calling `match`. `tshape` is called by `match` with a second argument taken from a `table` entry. `tshape` is also called by `reclaim` (2742), `setasg` (3508), and `adrput` (4562).

Because there is the call from `order`, it will be seen that there is an affinity between cookies and shapes, as is strongly suggested by the comments on lines 0394 to 0412.

```
        case FLD:
            if( shape & SFLD ){
                if( !flshape( p->left ) ) return(0);
                /* it is a FIELD shape; make side-effects */
                o = p->rval;
                fldsz = UPKFSZ(o);
# ifdef RTOLBYTES   >>>>>>>>>>>>>>>>>>>>>>>>>>
                fldshf = UPKFOFF(o);
# else                  xxxxxxxxxxxxxxxxxxxxxxxxxx
                fldshf = SZINT - fldsz - UPKFOFF(o);
# endif                 <<<<<<<<<<<<<<<<<<<<<<<<<<
                return(1);
                }
            return(0);

        case CCODES:
            return( shape&SCC );

        case REG:
            /* distinctions:
            SAREG   any scalar register
            STAREG  any temporary scalar register
            SBREG   any lvalue (index) register
            STBREG  any temporary lvalue register
            */
            mask = isbreg( p->rval ) ? SBREG : SAREG;
            if( istreg( p->rval ) && busy[p->rval]<=1 )
                mask |= mask==SAREG ? STAREG : STBREG;
            return( shape & mask );

        case OREG:
            return( shape & SOREG );

        case UNARY MUL:
            /* return STARNM or STARREG or 0 */
            return( shumul(p->left) & shape );

            }

        return(0);
        }
/* ---------------------------------------------------- */
```

`tshape` returns a true value if the node and the desired shape agree according to a rather complex set of criteria, and false otherwise. Note that a particular shape may include several distinct possibilities or alternatives.

2248: If the shape can be `SPECIAL`, then ...

2255: if the shape specifies one of -1, 0 or +1, check that `p` represents a constant, but not an address constant, and that its value is correct.

2262: `special` (4163) is a machine dependent routine which, on the PDP11, looks for character constants (`SCCON` (0344)) or positive integer constants (`SICON` (0346)). These are used as special cases in the tables for code optimization.

2266: If the shape is not important ... or if the calculation is for effect only ... then ok.

2268: If the shape is `INTEMP` (this will only occur via a call from `order` or `reclaim`), then call the machine dependent procedure `shtemp` (4187) to make the decision as to whether the shape is that of a temporary storage location.

2270: `SWADD` is the shape for a word address ...
... relevant for the Honeywell 6000.

2274: Now make the decision by considering the operator type first.

2277: The kind of straightforward decision you would expect.

2282: Fields again.

2284: `flshape` (4195) is a machine dependent routine that attempts to determine if the field is being applied to something reasonable. This is clearly a point of interaction between the contents of `table` and the machine dependent code.

2286: Unpack the field specifications and leave them in the global variables `fldsz` and `fldshf` to be picked up by `expand` later.

2317: `shumul` is a machine dependent procedure which determines the shape (`STARNM` or `STARREG` or neither of these) for subtrees whose root is a `UNARY MUL` operation.

It may be noted* that the set of possible shapes that will be recognized is quite circumscribed. Only at this point and at line 2284 is there an opportunity to match anything but simple nodes. For a machine such as the PDP11, this is somewhat restrictive. Thus, in the present version of the compiler, the meaning of `STARREG` has been extended to comprehend autoincrement and autodecrement addressing (see `shumul` (4147)).

8.5 `ttype` (2325)

This procedure is called from `match` with two arguments: `t`, a word extracted from a tree node defining an operand type, and `tword`, a type description extracted from an operator template in `table`. (Just to contribute to the general confusion, `tword` is of type `int` whereas `t` is of type `TWORD`, which happens to be defined as `unsigned int`.) The range of values for `tword` is the union of the values defined on lines 0417 to 0430.

For the PDP11, the actual values which occur in `table` (with their frequencies) are:

* Communication from Lee Benoy

```
ttype( t, tword ) TWORD t; {
	/* does the type t match tword */

	if( tword & TANY ) return(1);

	if( tdebug ){
	    printf( "ttype( %o, %o )\n", t, tword );
	    }
	if( ISPTR(t) && (tword&TPTRTO) ) {
	    do {
	        t = DECREF(t);
	    } while ( ISARY(t) );
	        /* arrays that are left are usually only
	           in structure references... */
	    return( ttype( t, tword&(~TPTRTO) ) );
	    }
	/* TPOINT means not simple */
	if( t != BTYPE(t) ) return( tword & TPOINT );
	if( tword & TPTRTO ) return(0);

	switch( t ){

	case CHAR:
	    return( tword & TCHAR );
	case SHORT:
	    return( tword & TSHORT );
	case STRTY:
	case UNIONTY:
	    return( tword & TSTRUCT );
	case INT:
	    return( tword & TINT );
	case UNSIGNED:
	    return( tword & TUNSIGNED );
	case USHORT:
	    return( tword & TUSHORT );
	case UCHAR:
	    return( tword & TUCHAR );
	case ULONG:
	    return( tword & TULONG );
	case LONG:
	    return( tword & TLONG );
	case FLOAT:
	    return( tword & TFLOAT );
	case DOUBLE:
	    return( tword & TDOUBLE );
	    }

	return(0);
	}
/* ================================================= */
```

```
69 TANY
64 TLONG|TULONG
40 TINT|TUNSIGNED|TPOINT
20 TDOUBLE
11 TINT
10 TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR
 8 TFLOAT
 8 TCHAR|TUCHAR
 5 TUNSIGNED|TPOINT
 5 TUCHAR
 4 TPOINT
 4 TINT|TUNSIGNED
 3 TPOINT|TINT|TUNSIGNED|TCHAR|TUCHAR
 2 TULONG
 2 TPOINT|TINT|TUNSIGNED
 2 TLONG
 2 TINT|TLONG|TULONG
 2 TCHAR
 1 TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR|TLONG|TULONG
 1 TINT|TUNSIGNED|TPOINT|TCHAR
 1 TINT|TUNSIGNED|TCHAR|TUCHAR|TPOINT
 1 TDOUBLE|TFLOAT
 1 TCHAR|TINT
```

The intention of `ttype` is to decide the suitability of the operator for the actual operand type.

2328: Simple enough.

2333: If the type is complex, and is a pointer to something, then ...

2334: discard the pointer attribute and any succeeding array attributes.

2339: This recursive call could be changed to an iteration ... change `tword` and go back to line 2330.

2342: If `t` does not represent a basic type, then it must be a pointer to something (which may be acceptable), or else a function, which will certainly not be acceptable at this point (always assuming that it can actually happen).

2343: If we reach this point, `t` represents a basic type. If you are still looking for a pointer type, forget it.

2345: All the cases in this `switch` statement are eminently straightforward. Surely something must be missing!

2372: The basic types not explicitly mentioned in the `switch` statement beginning at line 2345 are UNDEF, FARG, ENUMTY, and MOETY. Presumably their occurrence here would be a real surprise.

8.6 expand (2376)

This procedure is called by `match`, `cbgen`, `genargs` and `zzzcode` to expand the string passed as its third argument, in accordance with the cookie passed as its second argument, and under the control of the tree passed as its first argument.

`expand` will look for values set previously in `fldsz` and `fldshf`, when it is dealing with field operators.

```
expand( p, cookie, cp ) NODE *p;  register char *cp; {
        /* generate code by interpreting table entry */

        CONSZ val;

        for( ; *cp; ++cp ){
            switch( *cp ){

            default:
                PUTCHAR( *cp );
                continue;  /* this is the usual case... */

            case 'Z':  /* special machine dependent operations */
                zzzcode( p, *++cp );
                continue;

            case 'F':  /* this line deleted if FOREFF is active */
                if( cookie & FOREFF ) while( *++cp != '\n' ) { }
                continue;

            case 'S':  /* field size */
                printf( "%d", fldsz );
                continue;

            case 'H':  /* field shift */
                printf( "%d", fldshf );
                continue;

            case 'M':  /* field mask */
            case 'N':  /* complement of field mask */
                  val = 1;
                  val <<= fldsz;
                  --val;
                  val <<= fldshf;
                  adrcon( *cp=='M' ? val : ~val );
                  continue;

            case 'L':  /* output special label field */
                printf( "%d", p->label );
                continue;

            case 'O':  /* opcode string */
                hopcode( *++cp, p->op );
                continue;

            case 'B':  /* byte offset in word */
                val = getlr(p,*++cp)->lval;
                val = BYTEOFF(val);
                printf( CONFMT, val );
                continue;

            case 'C': /* for constant value only */
                conput( getlr( p, *++cp ) );
                continue;

            case 'I': /* in instruction */
                insput( getlr( p, *++cp ) );
                continue;

            case 'A': /* address of */
                adrput( getlr( p, *++cp ) );
                continue;

            case 'U': /* for upper half of address, only */
                upput( getlr( p, *++cp ) );
                continue;

                }

            }

        }
/* ------------------------------------------------- */
```

2381: Read the string defined by the third argument, examining each character.

2385: Most characters (in fact all but a few upper case characters) are copied directly to the standard output.

2388: `'Z'` is an escape to the machine-dependent routine `zzzcode` (4415), to provide special effects. The next character is passed to this procedure as an argument.

Most of the special characters have effects which are readily discerned from reading the source code. Special effects are achieved via the following set of machine dependent procedures, which are found in the file `local2.c`:

1. `zzzcode` does machine specific expansions.
2. `adrcon` emits a constant (actually a bit mask).
3. `hopcode` selects one of a set of instruction names.
4. `conput` emits a constant or a register name.
5. `insput` is a null procedure on the PDP11, but is used by the Honeywell 6000 compiler to generate register names.
6. `adrput` generates the symbolic address of an operand.
7. `upput` complements `adrput` for the other half of long types.

```
# include "mfile2"

# define TBUSY 01000
NODE resc[3];
int busy[REGSZ];
int maxa, mina, maxb, minb;

/* ---------------------------------------------------- */

allo0(){ /* free everything */

	register i;

	maxa = maxb = -1;
	mina = minb = 0;

	REGLOOP(i){
		busy[i] = 0;
		if( rstatus[i] & STAREG ){
			if( maxa<0 ) mina = i;
			maxa = i;
			}
		if( rstatus[i] & STBREG ){
			if( maxb<0 ) minb = i;
			maxb = i;
			}
		}
	}
/* ---------------------------------------------------- */

allchk(){
	/* check to ensure that all registers are free */

	register i;

	REGLOOP(i){
		if( istreg(i) && busy[i] ){
			cerror( "register allocation error");
			}
		}
	}
/* ---------------------------------------------------- */


allo( p, q ) NODE *p; struct optab *q; {

	register n, i, j;

	n = q->needs;
	i = 0;

	while( n & NACOUNT ){
		resc[i].op = REG;
		resc[i].rval = freereg( p, n&NAMASK );
		resc[i].lval = 0;
		resc[i].name[0] = '\0';
		n -= NAREG;
		++i;
		}

	while( n & NBCOUNT ){
		resc[i].op = REG;
		resc[i].rval = freereg( p, n&NBMASK );
		resc[i].lval = 0;
		resc[i].name[0] = '\0';
		n -= NBREG;
		++i;
		}

	if( n & NTMASK ){
		resc[i].op = OREG;
		resc[i].rval = TMPREG;
```

This file contains, for the most part, procedures that are concerned with allocating, freeing and checking the use of resources, especially the temporary registers.

`allo` (2493) is the basic procedure for assigning registers. It is passed, as parameters, references to a node and a `table` entry, and it attempts to obtain the necessary temporary registers and/or temporary space in the object time stack. `allo` calls `freereg` (2546) and `freetemp` (2647) to make these allocations. The second of these is fairly straightforward, but the task of the former, `freereg`, is rather more convoluted.

`freereg` (2546) relies on the advice of the procedure `usable` (2582), which decides whether a particular register may be used in a particular context. The latter's task is complicated by the possibilities of (a) register pairs; (b) sharing registers that are already "busy".

`reclaim` (2677) is concerned with restoring parts of the tree after code has been generated. The difficult case occurs when the result is in one or more registers that must be saved.

`recl2` (2839) is invoked by `reclaim` at each node of the tree which is being freed, to call `rfree`. `rfree` (2854) decrements the "busy" status count for temporary registers, while `rbusy` (2874) increments the "busy" status count for temporary registers.

There are two procedures at the end of the file that would be more at home with the other tree manipulation routines in the file `common`. They are not there because they are not needed in the first pass of the compiler. These are `ncopy` (2891) which copies the the contents of a node onto another node, and `tcopy` (2910) which makes a complete copy of a subtree.

Since processor register resources tend to be very individualistic, the raw data for the routines of this section are included within the machine dependent file, `local2.c`. Worthy of special note at this stage are:

1. `rstatus` (3717) which specifies, for each register, whether it is to be classed as type A or type B, and whether it may be used as a temporary register*.
2. `respref` is both the name of a two word structure defined on line 0524, and the name of an array of such structures, initialized beginning at line 3729. It is used to select the most appropriate one of a set of possible outcomes.

9.1 Declarations

The following are declared at the head of the file: a flag, `TBUSY`, two arrays and four variables. The role of the variables `maxa`, `mina`, `maxb` and `minb` is discussed in the next section.

`resc` is an array of type `NODE`. It features prominently during the actual code generation, when it is used to remember references to registers and temporary variables which are allocated in accordance with the requirements of particular templates.

`busy` is used to keep track of the commitments for particular temporary registers. In theory, the usage of `busy` is simple enough: when a reference to a temporary register is inserted in the tree, the corresponding element of `busy` is incremented; when the reference is removed, the element of `busy` is decremented. However it turns out that the actual details of the manipulation of `busy` are somewhat indirect (perhaps a better word would be "obscure"). This topic is taken up again in the last section of this chapter.

* On the PDP11, the floating point registers are of type B. On the VAX11/780, there are no type B registers.

```
                if( p->op == STCALL || p->op == STARG || p->op ==
                        UNARY STCALL || p->op == STASG ){
                    resc[i].lval = freetemp( (SZCHAR*p->stsize +
                        (SZINT-1))/SZINT );
                    }
                else {
                    resc[i].lval = freetemp( (n&NTMASK)/NTEMP );
                    }
                resc[i].name[0] = '\0';
                resc[i].lval = BITOOR(resc[i].lval);
                ++i;
                }

        /* turn off "temporarily busy" bit */

        REGLOOP(j){
                busy[j] &= ~TBUSY;
                }

        for( j=0; j<i; ++j ) if( resc[j].rval < 0 ) return(0);
        return(1);

        }
/* ------------------------------------------------- */

freereg( p, n ) NODE *p; {
        /* allocate a register of type n */
        /* p gives the type, if floating */

        register j;

        /* not general; means that only one register (the result)
                is OK for call */
        if( callop(p->op) ){
                j = callreg(p);
                if( usable( p, n, j ) ) return( j );
                /* have allocated callreg first */
                }
        j = p->rall & ~MUSTDO;
        if( j!=NOPREF && usable(p,n,j) ){ /* needed and not allocated */
                return( j );
                }
        if( n&NAMASK ){
                for( j=mina; j<=maxa; ++j ) if( rstatus[j]&STAREG ){
                        if( usable(p,n,j) ){
                                return( j );
                                }
                        }
                }
        else if( n &NBMASK ){
                for( j=minb; j<=maxb; ++j ) if( rstatus[j]&STBREG ){
                        if( usable(p,n,j) ){
                                return(j);
                                }
                        }
                }

        return( -1 );
        }
/* ------------------------------------------------- */

usable( p, n, r ) NODE *p; {
        /* decide if register r is usable in tree p to satisfy need n */

        /* checks, for the moment */
        if( !istreg(r) ) cerror("usable asked about nontemp register");

        if( busy[r] > 1 ) return(0);
        if( isbreg(r) ){
                if( n&NAMASK ) return(0);
```

9.2 allo0 (2458)

This procedure is called once by p2init during the initialization phase, to initialize the busy array and to set the values of maxa, mina, maxb and minb, to reflect the ranges of the two register types that should be searched to locate a temporary register. (For the PDP11, mina is 0, maxa is 4, minb is 9, and maxb is 12.

2465: REGLOOP (0530) is simply a shorthand for a for statement over the registers.

9.3 allchk (2479)

This procedure is called by main at line 1038, after each expression tree has been processed, and again by reclaim under the alias of callchk at line 2701. It looks to see if any temporary register is still marked as busy ... which is, at the time the procedure is called, a serious error.

9.4 allo (2493)

This procedure is called at line 2202, as the last conditional step in match, before expand is called to generate assembler code. allo builds a list in the array resc (2452), whose elements are of type NODE, for each resource required by the table entry.

2497: Extract the "needs" from the table entry.

For the PDP11, the actual values which can occur here are:

0	NAREG	NBREG
NAREG¦NASL	NAREG¦NASR	NAREG¦NASL¦NASR
NBREG¦NBSR	NTEMP	2*NTEMP
4*NTEMP	REWRITE	

2500: If any type A registers are needed, set values in the next available element of resc.

2502: The interesting part is done by freereg (2546).

2509: Do the same for type B registers.

2518: Request is for temporary stack space.

2520: TMPREG (0335) is defined as R5 for the PDP11. Note that temporaries in the object time stack are referenced relative to the frame pointer, R5, and not the stack pointer.

2523: For structures, the size in characters is stored in the node as the field stsize. This value, converted to integer units, and rounded up, is passed as the argument to freetemp (2647). The value which is returned and stored in the field lval is the offset relative to TMPREG needed to locate the space allocated by freetemp. Note that freetemp, which never fails to make an allocation, accepts an argument measured in words, and returns an offset measured in bits.

2530: BITOOR (0331) is a machine dependent operation which converts its argument from bits to the addressable unit of storage. For the PDP11 and the VAX11/780, BITOOR simply shifts right by three places.

2536: freereg turns on the TBUSY flag for any register that it allocates. Such bits are now turned off, whether the allocation is regarded as successful or not. (Whether code is about to be generated or not, the registers will be available next time allo is called.)

```
		}
	else {
		if( n & NBMASK ) return(0);
		}
	if( (n&NAMASK) && (szty(p->type) == 2) ){
		/* only do the pairing for real regs */
		if( r&01 ) return(0);
		if( !istreg(r+1) ) return( 0 );
		if( busy[r+1] > 1 ) return( 0 );
		if( busy[r] == 0 && busy[r+1] == 0  ||
		    busy[r+1] == 0 && shareit( p, r, n ) ||
		    busy[r] == 0 && shareit( p, r+1, n ) ){
			busy[r] |= TBUSY;
			busy[r+1] |= TBUSY;
		      return(1);
		      }
		else return(0);
		}
	if( busy[r] == 0 ) {
		busy[r] |= TBUSY;
		return(1);
		}

	/* busy[r] is 1: is there chance for sharing */
	return( shareit( p, r, n ) );

	}
/* ----------------------------------------------------- */

shareit( p, r, n ) NODE *p; {
	/* can we make register r available by sharing from p
	   given that the need is n */
	if( (n&(NASL|NBSL)) && ushare( p, 'L', r ) ) return(1);
	if( (n&(NASR|NBSR)) && ushare( p, 'R', r ) ) return(1);
	return(0);
	}
/* ----------------------------------------------------- */

ushare( p, f, r ) NODE *p; {
	/* can we find a register r to share on the left or right
		(as f=='L' or 'R', respectively) of p */
	p = getlr( p, f );
	if( p->op == UNARY MUL ) p = p->left;
	if( p->op == OREG ){
		if( R2TEST(p->rval) ){
			return( r==R2UPK1(p->rval) || r==R2UPK2(p->rval) );
			}
		else return( r == p->rval );
		}
	if( p->op == REG ){
		return(r==p->rval || (szty(p->type)==2 && r==p->rval+1));
		}
	return(0);
	}
/* ----------------------------------------------------- */

freetemp( k ){ /* allocate k integers worth of temp space */
	/* we also make the convention that, if the number of words
	/*     is more than 1,
	/* it must be aligned for storing doubles... */

# ifndef BACKTEMP
	int t;

	if( k>1 ){
		SETOFF( tmpoff, ALDOUBLE );
		}

	t = tmpoff;
	tmpoff += k*SZINT;
	if( tmpoff > maxoff ) maxoff = tmpoff;
```

2540: A final check is made to see if `allo` has been successful: for every relevant element of `resc`, is the register number stored in the `rval` field valid? (`freereg` will have returned −1 if it failed.)

Clearly, `allo` is more general than is needed by the PDP11: from the list of actual "needs" given above, `allo` is only ever called upon to allocate:

1. exactly one A register or register pair; or
2. exactly one B register or register pair; or
3. one block of stack storage; or
4. nothing

9.5 Free Registers

The next four procedures, `freereg`, `usable`, `shareit` and `ushare`, form a strict hierarchy, where each is called by, and only by, its predecessor, with the exception of `freereg`, which is called by `allo` at lines 2502, 2511.

9.5.1 `freereg` *(2546)* is called to allocate a free register. The first argument is a node pointer and the second indicates whether a type A or type B register is needed.

2554: If the operation is a "call" (`CALLFLG` set; see `callop` (0159) and `dope` (0724)) then take the value returned by the machine dependent routine `callreg` (4021). For the PDP11, this is either `R0` or `FR0`.

2556: Check if the register is "usable" (see later).

2559: Look at the value in the `rall` field of the node, but ignore the `MUSTDO` flag if it is set.

2560: If a definite register has been requested, and it is "usable", return the register number.

2563: Look for either a type A or a type B temporary register which is "usable" and return its value, else ...

2578: return −1 as an indication of failure.

9.5.2 `usable` *(2582)* is called by `freereg` to determine whether a given register is available to be used.

2588: Failure. The register is already committed more than once.

2589: Failure. This is a B register and you wanted an A, or vice versa.

2595: A pair of type A registers is required. They must be an even-odd pair, and both must be available, or potentially available (`shareit` (2620)).

2603: Mark the registers busy, and return.

2609: If the register is free, reserve the register and return.

2614: The register is booked, but there is still a chance. Look a little further.

9.5.3 `shareit` *(2620)* The arguments passed to this procedure are a re-ordering of the arguments of its parent, `usable`.

2623: If the template says that the left operand may be shared, call `ushare` to check the left operand.

```
	if( tmpoff-baseoff > maxtemp ) maxtemp = tmpoff-baseoff;
	return(t);

# else
	tmpoff += k*SZINT;
	if( k>1 ) {
		SETOFF( tmpoff, ALDOUBLE );
		}
	if( tmpoff > maxoff ) maxoff = tmpoff;
	if( tmpoff-baseoff > maxtemp ) maxtemp = tmpoff-baseoff;
	return( -tmpoff );
# endif
	}
/* ================================================ */

reclaim( p, rw, cookie ) NODE *p; {
	register NODE **qq;
	register NODE *q;
	register i;
	NODE *recres[5];
	struct respref *r;

	/* get back stuff */

	if( rdebug ){
		printf( "reclaim( %o, ", p );
		rwprint( rw );
		printf( ", " );
		prcook( cookie );
		printf( " )\n" );
		}

	if( rw == RNOP || ( p->op==FREE && rw==RNULL ) )
		return;  /* do nothing */

	walkf( p, recl2 );

	if( callop(p->op) ){
		/* check that all scratch regs are free */
		callchk(p); /* ordinarily, this is the same as allchk() */
		}

	if( rw == RNULL || (cookie&FOREFF) ){
		/* totally clobber, leaving nothing */
		tfree(p);
		return;
		}

	/* handle condition codes specially */

	if( (cookie & FORCC) && (rw&RESCC)) {
		/* result is CC register */
		tfree(p);
		p->op = CCODES;
		p->lval = 0;
		p->rval = 0;
		return;
		}

	/* locate results */

	qq = recres;

	if( rw&RLEFT) *qq++ = p->left;
	if( rw&RRIGHT ) *qq++ = p->right;
	if( rw&RESC1 ) *qq++ = &resc[0];
	if( rw&RESC2 ) *qq++ = &resc[1];
	if( rw&RESC3 ) *qq++ = &resc[2];

	if( qq == recres ){
		cerror( "illegal reclaim");
```

2624: If that failed, try the same with the right operand.

9.5.4 `ushare` *(2629)* When this procedure is called, by `shareit`, it is known that the register `r` (the third argument) has only a single commitment. After the operation on line 2632, `p` designates a subtree, whose result, so far as the current template is concerned, may be shared, i.e. its value is needed as data for the instruction execution, but may be destroyed by the end of the execution. The question to be answered is: does `r` designate a register which is appropriately located in the subtree designated by `p`?

2632: Descend the tree one level, if possible.

2633: Descend an additional level if the operator is a UNARY MUL.

2634: If the node is an OREG, is `r` either the base or displacement register?

2640: If the node is a REG, is `r` the register? Or could it be that the node denotes a register pair, and `r` is the other member of the pair?

9.6 `freetemp` (2647)

This procedure is called by `allo` (at line 2523 or 2527) to allocate temporary stack space. There are two distinct approaches, depending on whether stacks grow up or down. On the PDP11, they grow down.

2666: Increase `tmpoff` by the number of bits requested. (The request was in terms of words.)

2655: If more than one word was requested, align the allocated area on a double word boundary. This means rounding the value for `tmpoff` up. (This is a conservative strategy, which should nip most alignment problems in the bud.)

2670: Keep the values of `maxoff` and `maxtemp` current.

2672: Return the negative value of `tmpoff`. This will be used as an offset from R5 to find the beginning of the newly allocated temporary area.

9.7 `reclaim` (2677)

This procedure is called by `cbgen`, `cbranch`, `genargs`, `main`, `match`, `order` and `setasop`, to rewrite a subtree after code has been generated. The revised tree will reflect the values which will be generated by the newly emitted code. There are three arguments: a node pointer, directions for how the tree is to be rewritten, and the original "cookie" or set of alternative goals.

2681: Note the dynamic array allocation for `recres`, which is used in the reclamation of resources.

2694: The easy cases. For the PDP11, RNOP occurs with a template for STASG (5426) (structure assignment), and with templates for GOTO (lines 05464 to 5480). RNULL occurs for templates where `visit` (i.e. "cookie") is FORARG, so that the results will go into the stack.

2697: Walk the tree in preorder, and apply `recl2` (2839) at each node to "free" any registers in use.

2699: Check the "busy" states of temporary type A registers.

```
		}

	*qq = NIL;

	/* now, select the best result, based on the cookie */

	for( r=respref; r->cform; ++r ){
		if( cookie & r->cform ){
			for( qq=recres; (q= *qq) != NIL; ++qq ){
				if( tshape( q, r->mform ) ) goto gotit;
				}
			}
		}

	/* we can't do it; die */
	cerror( "cannot reclaim");

	gotit:

	if( p->op == STARG ) p = p->left;  /* STARGs are still STARGS */

	/* to make multi-register allocations work */
	q->type = p->type;
		/* maybe there is a better way! */

	q = tcopy(q);
	tfree(p);
	p->op = q->op;
	p->lval = q->lval;
	p->rval = q->rval;
	for( i=0; i<NCHNAM; ++i )
		p->name[i] = q->name[i];
	q->op = FREE;

	/* if the thing is in a register, adjust the type */

	switch( p->op ){

	case REG:
		if( p->type == CHAR || p->type == SHORT ) p->type = INT;
		else if( p->type == UCHAR || p->type == USHORT )
			p->type = UNSIGNED;
		else if( p->type == FLOAT ) p->type = DOUBLE;
		if( ! (p->rall & MUSTDO ) ) return;
			/* unless necessary, ignore it */
		i = p->rall & ~MUSTDO;
		if( i & NOPREF ) return;
		if( i != p->rval ){
			if( busy[i] || ( szty(p->type)==2 && busy[i+1] ) ){
				cerror( "faulty register move" );
				}
			rbusy( i, p->type );
			rfree( p->rval, p->type );
			rmove( i, p->rval, p->type );
			p->rval = i;
			}

	case OREG:
		if( R2TEST(p->rval) ){
			int r1, r2;
			r1 = R2UPK1(p->rval);
			r2 = R2UPK2(p->rval);
			if( (busy[r1]>1 && istreg(r1)) ||
					(busy[r2]>1 && istreg(r2)) ){
				cerror( "potential register overwrite" );
				}
			}
		else if( (busy[p->rval]>1) && istreg(p->rval) )
				cerror( "potential register overwrite");
		}
	}
/* ------------------------------------------------ */
```

2704: If the tree was evaluated "for effect", or if there is nothing to be saved, dismantle the subtree.

2712: If the cookie included `FORCC`, and the result is accessible via the current condition codes ... ok.

2720: If the cookie was `FORCC` alone, and we get here ... then die at line 2732.

2723: Make a list of resources that are candidates to replace the subtree denoted by `p`, as given by the template rewriting specifications.

2739: The problem here is to choose the most useful result from among the possible results, which are now listed in the array `recres`. A certain amount of leeway may be possible in some cases if the original "cookie" was not matched exactly. `respref` (03729) is a list of pairs (`cform`, `mform`), given in order of preference. If the "cookie" matches `cform`, see if one of the `recres` elements is acceptable on the basis of `mform`. The result does not have to match the "cookie" exactly provided it is close enough. For example, if the "cookie" was `INAREG`, then any addressable type will be acceptable, because it can be taken care of by the final call to `match` at line 1793 of `order`, if necessary.

2742: Quit as soon as a result is found that the shape of one of the alternative "cookies".

2752: Descend the tree one level, so that the `STARG` will not be thrown away by the code beginning at at line 2759.

2754: Operand type information was not stored in `resc` by `allo` (2493) earlier.

2758: Note that `tcopy` updates `busy` counts.

2771: Adjust the type of `REG` nodes, i.e. widen the value if necessary.

2780: The result is in the wrong register. Generate a register-to-register move.

2790: Only a test for compiler consistency.

9.8 `rwprint` (2806)

This procedure, which is invoked by `reclaim` at line 2688 for diagnostic printing, serves as a working definition for the set of values that `reclaim` can expect as its second argument.

9.9 `recl2` (2839)

This procedure is passed as the second argument to `walkf` (0688) by `reclaim` at line 2697. For each register reference in the subtree that is being freed, call `rfree` (2854) to update the corresponding element of `busy`.

9.10 `rfree` (2854)

If `r` is a temporary register, decrement `busy[r]` to reflect a use for `r` which is being given up. Take care of register pairs, as and when required, and be cautious about error situations. This procedure is called by `recl2`, and also by `reclaim` and `zzzcode` at lines 2785, 4589 respectively.

9.11 `rbusy` (2874)

This procedure implements the reverse operation to that performed by `rfree`. It is called by `eread` at line 1118, `reclaim` at line 2784, and `tcopy` (2910) (four times), and `zzzcode` at line 4592.

```

rwprint( rw ){ /* print rewriting rule */
	register i, flag;
	static char * rwnames[] = {

		"RLEFT",
		"RRIGHT",
		"RESC1",
		"RESC2",
		"RESC3",
		0,
		};

	if( rw == RNULL ){
		printf( "RNULL" );
		return;
		}

	if( rw == RNOP ){
		printf( "RNOP" );
		return;
		}

	flag = 0;
	for( i=0; rwnames[i]; ++i ){
		if( rw & (1<<i) ){
			if( flag ) printf( "|" );
			++flag;
			printf( rwnames[i] );
			}
		}
	}
/* ------------------------------------------------------------ */

recl2( p ) register NODE *p; {
	register r = p->rval;
	if( p->op == REG ) rfree( r, p->type );
	else if( p->op == OREG ) {
		if( R2TEST( r ) ) {
			rfree( R2UPK1( r ), PTR+INT );
			rfree( R2UPK2( r ), INT );
			}
		else {
			rfree( r, PTR+INT );
			}
		}
	}
/* ------------------------------------------------------------ */

rfree( r, t ) TWORD t; {
	/* mark register r free, if it is legal to do so */
	/* t is the type */

	if( rdebug ){
		printf( "rfree( %s ), size %d\n", rnames[r], szty(t) );
		}

	if( istreg(r) ){
		if( --busy[r] < 0 ) cerror( "register overfreed");
		if( szty(t) == 2 ){
			if( (r&01) || (istreg(r)^istreg(r+1)) )
				cerror( "illegal free" );
			if( --busy[r+1] < 0 )
				cerror( "register overfreed" );
			}
		}
	}
/* ------------------------------------------------------------ */
```

9.12 `ncopy` (2891)

This procedure is called by `delay1`, `delay2`, `order` and `tcopy` at lines 1223, 1267, 1610 and 2916 respectively. It copies the contents of one node (given as the second argument) onto another (the first argument). It is useful when one subtree must be replaced by another. Since it may be difficult to locate all existing references to the root of the first subtree, it is easier to copy the content of the root of the second subtree onto the element that was the root of the first subtree, and to abandon the element which was the root of the second subtree.

9.13 `tcopy` (2910)

If we ignore for the moment the code on lines 2918 through 2928, this procedure is an archetype for a recursive "tree copy" routine. The contents of individual nodes are copied by the call to `ncopy` at line 2916. `tcopy` is called by `delay2` at line 1264, by `order` at lines 1724, 1740, and by `reclaim`, `setasop` and `zzzcode` at lines 2758, 3452 and 4444 respectively.

2918: The code from here to line 2928 is almost identical to that on lines 2840 to 2850, except that `rfree` has been replaced by `rbusy`. Thus it will be seen that, as new copies of subtrees are made, for each register reference which is encountered, `rbusy` is called to increment the appropriate element of `busy`, if the register is a temporary register.

9.14 Keeping `busy`

As mentioned at the beginning of this chapter, keeping track of the movements of the elements of `busy` in this program is not a straightforward task. Moreover since register allocation is such a central problem in the whole task of code generation, any failure in the mechanism for manipulating `busy` could have serious consequences. Since part of this mechanism resides in the machine-dependent parts of the compiler, new implementers should take care. A review of operations on `busy` needs to consider the following points:

1. nodes of type `REG` are recognized by `eread` when expression trees are read in from the intermediate file. Temporary registers should not appear here, but `rbusy` is called anyway (line 1118).
2. Since `eread` does not recognize `OREG` nodes, it can be assumed that these will not be present in the initial trees, or, if present, do not use a temporary register.
3. `OREG` nodes are generated by `oreg2` (1988), which uses `tfree` to dismantle a subtree and replace it by a single node. `tfree` (0675) does not call `rfree`, and so the array `busy` is not altered during this operation.
4. When trees are copied by `tcopy`, as an important side-effect, the `busy` counts for temporary registers are increased.
5. When trees are dismantled by `reclaim`, `busy` counts for temporary registers are decreased.
6. The most important place where `busy` counts are incremented is not at all obvious: it occurs as a side effect of the code on lines 2758 through 2765 in `reclaim`.

The complexity of a complete verification of the program's manipulations of `busy` is sufficiently daunting that the present writer has not attempted it. This is, of course, not to say that the code is incorrect. However, a complete check would need to examine all sections of code which rewrite the trees or the contents of nodes to ensure that references to registers are not being created or destroyed under obscure circumstances. It seems to the present writer that this aspect of the present program should not be considered one of its more enduring features.

Steve Johnson has pointed out that the use of `allchk` ensures that disasters in this area can't spread, and also that register sharing is one aspect of the compiler over which he labored long, and successfully! The present code does tackle the problem in a machine-independent way, which is an achievement in itself.

```

rbusy(r,t) TWORD t; {
	/* mark register r busy */
	/* t is the type */

	if( rdebug ){
		printf( "rbusy( %s ), size %d\n", rnames[r], szty(t) );
		}

	if( istreg(r) ) ++busy[r];
	if( szty(t) == 2 ){
		if( istreg(r+1) ) ++busy[r+1];
		if( (r&01) || (istreg(r)^istreg(r+1)) )
				cerror( "illegal register pair freed" );
		}
	}
/* ================================================== */

ncopy( q, p ) NODE *p, *q; {
	/* copy the contents of p into q, without any feeling for
	   the contents */
	/* this code assume that copying rval and lval does the job;
	   in general, it might be necessary to special case the
	   operator types */
	register i;

	q->op = p->op;
	q->rall = p->rall;
	q->type = p->type;
	q->lval = p->lval;
	q->rval = p->rval;
	for( i=0; i<NCHNAM; ++i ) q->name[i]  = p->name[i];

	}
/* -------------------------------------------------- */

NODE *
tcopy( p ) register NODE *p; {
	/* make a fresh copy of p */

	register NODE *q;
	register r;

	ncopy( q=talloc(), p );

	r = p->rval;
	if( p->op == REG ) rbusy( r, p->type );
	else if( p->op == OREG ) {
		if( R2TEST(r) ){
			rbusy( R2UPK1(r), PTR+INT );
			rbusy( R2UPK2(r), INT );
			}
		else {
			rbusy( r, PTR+INT );
			}
		}

	switch( optype(q->op) ){

	case BITYPE:
		q->right = tcopy(p->right);
	case UTYPE:
		q->left = tcopy(p->left);
		}

	return(q);
	}
/* -------------------------------------------------- */
```

The file `order.c` contains procedures which, to a greater or less extent, are machine-dependent. As the name suggests, many of these are associated with the procedure `order`, and represent sections of code which might naturally occur in-line in that procedure. However, in the absence of better mechanisms for building program families, these machine-dependent sequences have been exorcised and made into the free-standing procedures that appear here.

Of the nineteen procedures in this file, there are three

`offstar`	called by `genargs`, `order`, `setasg`, `setasop` and `setbin`
`getlab`	called by `cbgen`, `cbranch` and `order`
`deflab`	called by `cbgen`, `cbranch`, `order` and `zzzcode`

which lay some claim to being of general usefulness. There are two other procedures

`rallo`	called by `order`, `setasop` and `mkrall`
`stoasg`	called by `store`

which are invoked more than once by procedures external to this file. Most of the remaining procedures, namely

`deltest`	called by	`delay2`	(1261)
`mkadrs`	called by	`store`	(1383)
`sucomp`	called by	`canon`	(1319)
`setincr`	called by	`order`	(1713)
`setstr`	called by	`order`	(1732)
`setasop`	called by	`order`	(1736)
`setasg`	called by	`order`	(1754)
`setbin`	called by	`order`	(1759)
`notoff`	called by	`oreg2`	(2065)
`genargs`	called by	`gencall`	(4041)
`argsize`	called by	`gencall`	(4037)

consist of straight line code and are called exactly once. Amongst these, only `sucomp` should undoubtedly be a separate procedure on its own merits. The last two procedures, `genargs` and `argsize`, are really out of place, and should be moved to be with their "parent", `gencall`, into the file `local2.c`.

Finally there is a small set of procedures that are only referenced from within this file, i.e. they are not referenced from the machine-independent parts of the program, and hence could conceivably not appear in some other implementations:

`zum`	called by `sucomp`
`mkrall`	calls, and called, by `rallo`
`niceuty`	called by `setbin`

The single variable declared at the head of this file, `fltused`, is used as a flag to signal the occurrence of floating point operations. It is incremented by `rallo` at line 3022, whenever it

```
# include "mfile2"

int fltused = 0;

/* -------------------------------------------------- */

deltest( p ) register NODE *p; {
	/* should we delay the INCR or DECR operation p */
	if( p->op == INCR && p->left->op == REG &&
			spsz( p->left->type, p->right->lval ) ){
		/* STARREG */
		return( 0 );
		}
	p = p->left;
	if( p->op == UNARY MUL ) p = p->left;
	return( p->op == NAME || p->op == OREG || p->op == REG );
	}
/* -------------------------------------------------- */

stoasg( p, o ) register NODE *p; {
	/* should the assignment op p be stored,
	   given that it lies as the right operand of o
	   (or the left, if o==UNARY MUL) */
	return( shltype(p->left->op, p->left ) );
	}
/* -------------------------------------------------- */

mkadrs(p) register NODE *p; {
	register o;

	o = p->op;

	if( asgop(o) ){
		if( p->left->su >= p->right->su ){
			if( p->left->op == UNARY MUL ){
				if( p->left->su > 0 )
					SETSTO( p->left->left, INTEMP );
				else {
					if( p->right->su > 0 )
						SETSTO( p->right, INTEMP );
					else cerror(
					"store finds both sides trivial" );
					}
				}
			else if( p->left->op == FLD &&
					p->left->left->op == UNARY MUL ){
				SETSTO( p->left->left->left, INTEMP );
				}
			else { /* should be only structure assignment */
				SETSTO( p->left, INTEMP );
				}
			}
		else SETSTO( p->right, INTEMP );
		}
	else {
		if( p->left->su > p->right->su ){
			SETSTO( p->left, INTEMP );
			}
		else {
			SETSTO( p->right, INTEMP );
			}
		}
	}
/* -------------------------------------------------- */
```

encounters a node of type FLOAT or DOUBLE, and interrogated by eobl2 (3755), which then passes the information to the assembler. (This is totally oriented towards the PDP11.)

10.1 deltest (2947)

This procedure is called by delay2 at line 1261 to determine whether INCR and DECR operations may be executed after the main effects of the expression have been realized. If the operation is delayed, a copy of the subtree of the INCR or DECR operator is made, and a reference to it is stored in the array deltrees. The main tree can then be simplified, and, in particular, the INCR or DECR operator can be removed.

deltest for the PDP11 returns the answer "do not delay" if the incrementation can be performed naturally by the hardware using autoincrement addressing modes*, or if the operand is not directly addressable.

10.2 stoasg (2960)

This procedure is called by store twice, at lines 1345 and 1380. In neither case does the calling procedure expect a value to be returned. shltype (4141) may call shumul (4147), which may call spsz (4096), but since none of these has any side-effect, it can be seen that stoasg, at least for the PDP11, is harmless (and should be null, as it is for the VAX11/780).

10.3 mkadrs (2968)

mkadrs is called by store, when the latter knows that some intermediate result will have to be stored temporarily in the stack. The question then becomes "from which subtree will this result come?". The decision is frequently made by mkadrs.

Unfortunately for the reader, the logic of this procedure is inverted, with the most complicated situation being given first, and the easy cases being left until later.

2995: We are not dealing with an assignment operator so ...

2996: if the left side looks harder, ...

2997: do it, otherwise ...

2999: do the right side first.

2993: We are dealing with an assignment operator, and the right hand side looks the harder, so do it first.

2985: We are dealing with an assignment operator, but the operator in the left subtree is not a UNARY MUL. Perhaps it is a FLD pointing at a UNARY MUL pointing at ... If so, store ... !

2975: We are dealing with an assignment operator; the left side is at least as demanding of temporary registers as the right side; the left subtree has a UNARY MUL root ...

2976: How bad really is the left hand side? If it needs at least one register, get the address (subtree of the UNARY MUL) into temporary storage. (Seems a fairly conservative response.)

---

* Lee Benoy has pointed out that, with a very high probability, the parent of the INCR node will be a UNARY MUL. If it is not, then it would in fact be better to delay.

```
rallo( p, down ) register NODE *p; {
	/* do register allocation */
	register o, type, down1, down2, ty;

	if( radebug ) printf( "rallo( %o, %o )\n", p, down );

	down2 = NOPREF;
	p->rall = down;
	down1 = ( down &= ~MUSTDO );

	ty = optype( o = p->op );
	type = p->type;


	if( type == DOUBLE || type == FLOAT ){
		if( o == FORCE ) down1 = FR0|MUSTDO;
		++fltused;
		}
	else switch( o ) {
	case ASSIGN:
		down1 = NOPREF;
		down2 = down;
		break;

	case ASG MUL:
	case ASG DIV:
	case ASG MOD:
		/* keep the addresses out of the hair of (r0,r1) */
		if(fregs == 2 ){
			/* lhs in (r0,r1), nothing else matters */
			down1 = R1|MUSTDO;
			down2 = NOPREF;
			break;
			}
		/* at least 3 regs free */
		/* compute lhs in (r0,r1), address of left in r2 */
		p->left->rall = R1|MUSTDO;
		mkrall( p->left, R2|MUSTDO );
		/* now, deal with right */
		if( fregs == 3 ) rallo( p->right, NOPREF );
		else {
			/* put address of long or value here */
			p->right->rall = R3|MUSTDO;
			mkrall( p->right, R3|MUSTDO );
			}
		return;

	case MUL:
	case DIV:
	case MOD:
		rallo( p->left, R1|MUSTDO );

		if( fregs == 2 ){
			rallo( p->right, NOPREF );
			return;
			}
		/* compute addresses, stay away from (r0,r1) */
		p->right->rall = (fregs==3) ? R2|MUSTDO : R3|MUSTDO ;
		mkrall( p->right, R2|MUSTDO );
		return;

	case CALL:
	case STASG:
	case EQ:
	case NE:
	case GT:
	case GE:
	case LT:
	case LE:
```

2980: If we get here, it should be a miracle or something*, since:

```
p->left->su >= p->right->su
p->left->su == 0
p->right->su > 0
```

2990: When further inspiration fails ... do this.

Since there is no way out of this procedure without executing a statement of the form

```
SETSTO( ..., INTEMP );
```

and since `SETSTO` is a macro which stores values for `stotree` and `stocook`, this procedure could be improved space-wise slightly by setting `stocook` upon entry, and changing the references to `SETSTO` to assignments to `stotree`.

10.4 `rallo` (3006)

This procedure is called from `order` twice and once from `setasop` (3398), which is really an in-line segment of `order`. `rallo` also has a recursive relationship with its alter ego `mkrall`.

Notwithstanding the comment on line 3007, `rallo` does not perform register allocation explicitly. This is done by `freereg` (2546) (called by `allo`) when the time for code generation actually arrives. The task of `rallo` is to set values of the `rall` field of tree nodes. These values, unless they are flagged as MUSTDO, constitute advice, not orders, to `freereg`. (The `rall` value is also observed by `reclaim` at line 2778, which may generate a "register-to-register" move, if perchance the result has been forced into the wrong register.)

The basic strategy of `rallo` is to perform a pre-order walk of the subtree, setting the `rall` field of each node as appropriate, in order to direct the results of the calculation into the location specified by the `rall` value of the root node. (This advice may range from NOPREF to something very specific.) The nodes are not treated exactly alike since certain links may be traversed via calls to `mkrall` rather than to `rallo`.

The call from `order` (1539) is executed upon the initial entry to `order` and at the beginning of every subsequent iteration. The other calls, from `order` at line 1745 and `setasop` at line 3473, are made after the tree has been rewritten and immediately before a recursive call on `order`.

3012: Set out to:

1. Tell your right descendent nothing.
2. Do as you were told by your parent.
3. Be a little less strict with your left descendent.

`down1` and `down2` are passed as arguments to recursive calls to `rallo` at lines 3087 and 3088 for the left and right subtrees respectively.

3020: If the data type is single or double precision floating point, and the operator is `FORCE`, give the left descendent strict instructions.

3025: With `ASSIGN` operations, the result will be where the right subtree leaves its result, so plan accordingly.

3030: Multiplication and division with assignment. If a register pair is needed, this will be R0 and R1. The preferred strategy is to obtain, in order of importance, R1 for the value of the left operand, R2 for any register used to address the left operand, and R3 for the value of the right operand. If, when code generation time draws near, a spare register has to be found (as specified in the template), this will be R0.

* Lee Benoy comments: "Very interesting, the tangle one gets oneself into ... "

```
	case NOT:
	case ANDAND:
	case OROR:
		down1 = NOPREF;
		break;

	case FORCE:
		down1 = R0|MUSTDO;
		break;

		}

	if( ty != LTYPE ) rallo( p->left, down1 );
	if( ty == BITYPE ) rallo( p->right, down2 );

	}
/* ---------------------------------------------------- */

mkrall( p, r ) register NODE *p; {
	/* insure that the use of p gets done with register r;
	/* in effect, simulate offstar */

	if( p->op == FLD ){
		p->left->rall = p->rall;
		p = p->left;
		}
	if( p->op != UNARY MUL ) return;  /* no more to do */
	p = p->left;
	if( p->op == UNARY MUL ){
		p->rall = r;
		p = p->left;
		}
	if( p->op == PLUS && p->right->op == ICON ){
		p->rall = r;
		p = p->left;
		}
	rallo( p, r );
	}
/* ==================================================== */

# define max(x,y) ((x)<(y)?(y):(x))
# define min(x,y) ((x)<(y)?(x):(y))

# define ZCHAR 01
# define ZLONG 02
# define ZFLOAT 04

sucomp( p ) register NODE *p; {

	/* set the su field in the node to the sethi-ullman
	   number, or local equivalent */

	register o, ty, sul, sur;
	register nr;

	ty = optype( o=p->op);
	nr = szty( p->type );
	p->su = 0;

	if( ty == LTYPE ) {
		if( p->type==FLOAT ) p->su = 1;
		return;
		}
	else if( ty == UTYPE ){
		switch( o ) {
		case UNARY CALL:
		case UNARY STCALL:
			p->su = fregs;  /* all regs needed */
			return;

```

Note that whatever happens, all temporary registers will be used, and that this is relevant to the calculation of the SU numbers.

3036: With only two free registers, force the result of the left subtree into R1. Although no preference is being expressed, the result of the right subtree is going to end up in a temporary stack location.

3042: If the example of other cases in this `switch` statement were followed, a `break` would occur here, leading to a recursive call to `rallo` at line 3087. Instead the code reaches down explicitly into the root node of the left subtree and "fixes" it. `mkrall` is then called, and it does not touch the root of the tree it is passed. However it goes on and propagates, to a limited extent, the value it receives as an argument into some of the nodes further down.

3053: Regular multiplication and division. The left operand must go into R1, and the result will appear in either R0 or R1.

3062: If extra registers are available, arrange for the value of the right operand to be placed in R2 or R3, and any value needed in the calculation of the right operand, into R2.

3067: The operators listed here do not impose any preference for where the result of either tree will be left. The conditional operators expect to obtain their data from the condition codes.

3081: The significance of the FORCE operator is manifest at this point. (It would be tidier if the code on lines 3020 to 3023 were moved to here.)

10.5 `mkrall` (3093)

This procedure, which is called by `rallo` at lines 3043, 3049 and 3064, is similar in intent to `rallo` in that it sets `rall` values for nodes in the subtree designated by its first argument. The second argument is a value which may be forced into the `rall` field of nodes in the left subtree in certain cases. The general intent is to get the subtree into a form that may be converted into an OREG that will use the register designated by `r`.

```
		case UNARY MUL:
			if( shumul( p->left ) ) return;

		default:
			p->su = max( p->left->su, nr);
			return;
			}
		}


	/* If rhs needs n, lhs needs m, regular su computation */
	sul = p->left->su;
	sur = p->right->su;
	if( o == ASSIGN ){
		asop:  /* also used for +=, etc., to memory */
		if( sul==0 ){
			/* don't need to worry about the left side */
			p->su = max( sur, nr );
			}
		else {
			/* right, left address, op */
			if( sur == 0 ){
	/* just get the lhs address into a register, and mov */
	/* the `nr' covers the case where value is in reg afterwards */
				p->su = max( sul, nr );
				}
			else {
				/* right, left address, op */
				p->su = max( sur, nr+sul );
				}
			}
		return;
		}
	if( o == CALL || o == STCALL ){
		/* in effect, takes all free registers */
		p->su = fregs;
		return;
		}
	if( o == STASG ){
		/* right, then left */
		p->su = max( max( sul+nr, sur), fregs );
		return;
		}
	if( logop(o) ){
		/* do the harder side, then the easier side,
		/*  into registers */
		/* left then right, max(sul,sur+nr) */
		/* right then left, max(sur,sul+nr) */
		/* to hold both sides in regs: nr+nr */
		nr = szty( p->left->type );
		sul = zum( p->left, ZLONG|ZCHAR|ZFLOAT );
		sur = zum( p->right, ZLONG|ZCHAR|ZFLOAT );
		p->su = min( max(sul,sur+nr), max(sur,sul+nr) );
		return;
		}
	if( asgop(o) ){
		/* computed by doing right, doing left address,
			doing left, op, and store */
		switch( o ) {
		case INCR:
		case DECR:
			/* do as binary op */
			break;

		case ASG DIV:
		case ASG MOD:
		case ASG MUL:
			if( p->type!=FLOAT && p->type!=DOUBLE )
				nr = fregs;
			goto gencase;
```

The Sethi-Ullman numbers estimate the number of processor registers that will be required to obtain or contain the value calculated for a particular subtree. The estimation of these numbers before code generation is attempted, together with the use of these numbers in choosing the strategy for code generation, constitutes one of the novel features of the Portable C compiler.

The original theory (Ravi Sethi and J.D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions", *Journal of the ACM*, Vol.17, No.4, October 1970, pp.715-728.) relates to the case where resources are of a single, uniform type, namely word registers, and where binary operators can combine the contents of two registers, or of a register and a memory location, and leave the result in a register or memory location. Let `p` be a node that has left and right descendents `l` and `r`, and let each of `sup`, `sul` and `sur` denote the register requirement, or *SU number*, for each of the subtrees whose root nodes are `p`, `l` and `r` respectively. Then the basic result is that `sup` is defined recursively by

$$\mathtt{sup} = \mathit{max}\{\ \mathtt{tp},\ \mathtt{sul},\ \mathtt{sur},\ \mathit{min}\{\ \mathtt{sul} + \mathtt{tr},\ \mathtt{sur} + \mathtt{tl}\ \}\ \}$$

Here `tp`, `tl` and `tr` denote the number of registers to store the result calculated by each of the subtrees whose root nodes are `p`, `l` and `r`, respectively. In the case considered by Sethi and Ullman, `tp` is always one, except for leaf nodes representing values stored in main memory, for which the value is zero.

An alternative formulation of the above expression is

$$\mathtt{sup} = \mathit{min}\{\ \mathit{max}\{\ \mathtt{tp},\ \mathtt{sul},\ \mathtt{sur} + \mathtt{tl}\ \},\ \mathit{max}\{\ \mathtt{tp},\ \mathtt{sur},\ \mathtt{sul} + \mathtt{tr}\ \}\ \}$$

which reduces, if `tp`, `tl` and `tr` all have the same value, `nr`, to

$$\mathtt{sup} = \mathit{min}\{\ \mathit{max}\{\ \mathtt{sul},\ \mathtt{sur} + \mathtt{nr}\ \},\ \mathit{max}\{\ \mathtt{sur},\ \mathtt{sul} + \mathtt{nr}\ \}\ \}$$

The term *max* { `sul sur + nr` } in the above formula represents the number of registers needed if the expression is evaluated right-to-left. If `sul` is zero, the formula reduces further to just

$$\mathtt{sup} = \mathit{min}\{\ \mathtt{sur} + \mathtt{nr},\ \mathit{max}\{\ \mathtt{sur},\ \mathtt{nr}\ \}\ \} = \mathit{max}\{\ \mathtt{sur},\ \mathtt{nr}\ \}$$

The theory is not directly applicable in practice for several reasons. The C language features many assignment operators which were not considered originally, and for the PDP11, the following must also be considered:

1. Some operators (notably multiply and divide) may require a pair of consecutive registers to store their result (i.e. `tp == 2`).
2. For some operators such as ASG MUL, it is desirable, if not absolutely essential, to get both the left operand value and the left operand address into registers simultaneously.
3. Floating point calculations use a separate set of registers (usually not in short supply).
4. The results of some calculations may appear in the condition code bits of the processor status word.
5. The result from a function call is always left in `R0` or `FR0`.

The procedure `sucomp` (3122) is used to calculate a value for each node of the tree using a modified version of the Sethi-Ullman algorithm. The modifications are machine dependent.

```

		case ASG PLUS:
		case ASG MINUS:
		case ASG AND:  /* really bic */
		case ASG OR:
			if( p->type == INT || p->type == UNSIGNED ||
					ISPTR(p->type) ) goto asop;

	gencase:
		default:
			sur = zum( p->right, ZCHAR|ZLONG|ZFLOAT );
			if( sur == 0 ){ /* easy case: if addressable,
				do left value, op, store */
				if( sul == 0 ) p->su = nr;
				/* harder: left adr, val, op, store */
				else p->su = max( sul, nr+1 );
				}
			else {
			/* do right, left adr, left value, op, store */
				if( sul == 0 ){
				/* right, left value, op, store */
					p->su = max( sur, nr+nr );
					}
				else {
					p->su = max(sur, max(sul+nr, 1+nr+nr));
					}
				}
			return;
			}
		}

	switch( o ){
	case ANDAND:
	case OROR:
	case QUEST:
	case COLON:
	case COMOP:
		p->su = max( max(sul,sur), nr);
		return;
		}

	if( ( o==DIV || o==MOD || o==MUL )
	    && p->type!=FLOAT && p->type!=DOUBLE ) nr = fregs;
	if( o==PLUS || o==MUL || o==OR || o==ER ){
		/* AND is ruined by the hardware */
		/* permute: get the harder on the left */

		register rt, lt;

		/* if ... don't do it! */
		if( istnode( p->left ) || sul > sur ) goto noswap;

		/* look for a funny type on the left, one on the right */
		lt = p->left->type;
		rt = p->right->type;

		if( rt == FLOAT && lt == DOUBLE ) goto swap;

		if( (rt==CHAR||rt==UCHAR) &&
			(lt==INT||lt==UNSIGNED||ISPTR(lt)) ) goto swap;

		if( lt==LONG || lt==ULONG ){
			if( rt==LONG || rt==ULONG ){
				/* if one is a STARNM, swap */
				if( p->left->op == UNARY MUL && sul==0 )
					goto noswap;
				if( p->right->op == UNARY MUL &&
					p->left->op != UNARY MUL ) goto swap;
				goto noswap;
				}
```

and may over-estimate the SU numbers in certain heuristically determined situations. By occasionally over-estimating, but never under-estimating, the register requirements to evaluate each subtree, the results calculated by `sucomp` provide a safe basis upon which to generate code for the subtree, while avoiding the problem of running out of temporary registers unexpectedly.

Because the calculation of the SU numbers is performed independently of the code generation, there is a valuable built-in check on compiler consistency, *provided* the strategies followed by `allo` and `rallo`, by `order` and the "`set`" procedures, and by `sucomp` are all consistent and compatible.

11.1 `sucomp` (3122)

This procedure is called by `canon` at line 1319, for each node visited during an endorder ("bottom-up") traversal of the expression tree. The PDP11 version of this procedure has been made rather more complex than some of the other versions because of the problems of dealing with long (i.e. two word) integers.

3130: Set `ty`, `nr`, `o` and `p->su`.

3135: A type A register is needed for addressing the operand.

3146: If the shape of the subtree defined by this node is either `STARNM` or `STARREG`, leave `p->su` as zero.

3154: All operators considered after this point are binary.

3172: Right-to-left evaluation is needed.

3179: Don't try to leave values in temporary registers during procedure calls.

3185: Why isn't this just *fregs*?

3195: `zum` (3318) ensures that the SU values associated with certain operand types will not fall below a minimum threshold.

3213: Grab all available temporary registers. (See the earlier discussion for line 3030.)

3230: The present case is not covered officially by the theory because there is a need to have the address of the left operand and its value in registers simultaneously.

3239: The third part of this expression, `1 + nr + nr`, corresponds to the case where both operands plus the address of the left operand, are brought into registers simultaneously.

3252: Intermediate results during the evaluation of logical expressions live in the the condition code bits of the processor status word, and do not require a register.

3259: See the comments later, in Chapter 13, for `optim2`.

3265: `istnode` is defined at line 0528 and checks whether the node is a `REG`, and if so, whether it involves a temporary register.

3271: Starting here, investigate the possibility of interchanging the right and left subtrees.

3312: It is necessary to get both operands into a register before the operation. Hence the subexpression for `nr + nr`.

```
                        else if( p->left->op == UNARY MUL && sul == 0 )
                                goto noswap;
                        else  /* put long on right, unless STARNM */
                                goto swap;
                        }

                /* we are finished with the type stuff now; if one
                   is addressable, put it on the right */
                if( sul == 0 && sur != 0 ){

                        NODE *s;
                        int ssu;

                swap:
                        ssu = sul;  sul = sur; sur = ssu;
                        s = p->left;  p->left = p->right; p->right = s;
                        }
                }
        noswap:

        sur = zum( p->right, ZCHAR|ZLONG|ZFLOAT );
        if( sur == 0 ){
                /* get left value into a register, do op */
                p->su = max( nr, sul );
                }
        else {
                /* do harder into a register, then easier */
                p->su = max( nr+nr, min( max( sul, nr+sur ),
                                max( sur, nr+sul ) ) );
                }
        }
/* ------------------------------------------------- */

zum( p, zap ) register NODE *p; {
        /* zap Sethi-Ullman number for chars, longs, floats */
        /* in the case of longs, only STARNM's are zapped */
        /* ZCHAR, ZLONG, ZFLOAT are used to select the zapping */

        register su;

        su = p->su;

        switch( p->type ){

        case CHAR:
        case UCHAR:
                if( !(zap&ZCHAR) ) break;
                if( su == 0 ) p->su = su = 1;
                break;

        case LONG:
        case ULONG:
                if( !(zap&ZLONG) ) break;
                if( p->op == UNARY MUL && su == 0 ) p->su = su = 2;
                break;

        case FLOAT:
                if( !(zap&ZFLOAT) ) break;
                if( su == 0 ) p->su = su = 1;

                }

        return( su );
        }
/* ================================================= */
```

It is the fate of `sucomp` to be written, and then rewritten and refined several times during the development of a new version of the Portable C compiler, as register allocation bugs are uncovered, and ways of improving the code generated in certain cases are discovered. Under these circumstances, it is not surprising that that the code might become a little ragged around the edges. In the present version of `sucomp`, there are a long set of tests plus two separate `switch` statements all keyed on the node operator type `o`. This seems to be less than optimal, and an overhaul to the structure of this procedure, in particular to utilize a single large `switch`, would seem to be now due. (Since the VAX11/780 version of the Portable C compiler preserves a similar structure for `sucomp`, perhaps this should be taken as a word of advice to future implementers.)

11.2 `zum` (3318)

This procedure is called only by `sucomp` at lines 3195, 3196, 3225 and 3305. It ensures that the SU numbers associated with nodes for certain operand types will never fall below certain thresholds. This procedure is extremely machine-oriented, and has no analog in other versions of the compiler.

In all four calls, the second argument is `ZCHAR|ZLONG|ZFLOAT`, so it, together with lines 3331, 3337 and 3342 represent surplus baggage. A test for `su==0` at the beginning of `zum` would also be helpful.

```
int crslab = 10000;

getlab(){
	return( crslab++ );
	}
/* ---------------------------------------------------- */

deflab( l ){
	printf( "L%d:\n", l );
	}
/* ==================================================== */

offstar( p ) register NODE *p; {
	/* handle indirections */

	if( p->op == UNARY MUL ) p = p->left;

	if( p->op == PLUS || p->op == MINUS ){
		if( p->right->op == ICON ){
			order( p->left , INTAREG|INAREG );
			return;
			}
		}
	order( p, INTAREG|INAREG );
	}
/* ---------------------------------------------------- */

setincr( p ) NODE *p; {
	return( 0 );	/* for the moment, don't bother */
	}
/* ---------------------------------------------------- */

setstr( p ) register NODE *p; { /* structure assignment */
	if( p->right->op != REG ){
		order( p->right, INTAREG );
		return(1);
		}
	p = p->left;
	if( p->op != NAME && p->op != OREG ){
		if( p->op != UNARY MUL ) cerror( "bad setstr" );
		order( p->left, INTAREG );
		return( 1 );
		}
	return( 0 );
	}
/* ---------------------------------------------------- */

setasop( p ) register NODE *p; {
	/* setup for =ops */
	register sul, sur;
	register NODE *q, *p2;

	sul = p->left->su;
	sur = p->right->su;

	switch( p->op ){

	case ASG PLUS:
	case ASG OR:
	case ASG MINUS:
		if( p->type != INT && p->type != UNSIGNED &&
				!ISPTR(p->type) ) break;
		if( p->right->type == CHAR || p->right->type == UCHAR ){
			order( p->right, INAREG );
			return( 1 );
			}
		break;

```

This chapter covers the third and final part of the machine-dependent file `order.c`. The first two procedures, `getlab` and `deflab`, are "one-liners" concerned with the generation of labels. The remaining procedures derive, directly or indirectly, from the procedure `order`.

12.1 `getlab` (3353)

Define a new numeric label value.

12.2 `deflab` (3358)

Output an assembler statement declaring a label (character 'L' followed by a decimal integer).

12.3 `offstar` (3363)

This procedure is a little more general than most, and it is called from fourteen different locations in `order` and its minions, `setasg`, `setasop`, `setbin` and `genargs`. In each case when `offstar` is called, the parent node is an operator of type UNARY MUL, so that the subtree that is passed to `offstar` will return a result that is an address.

The function of `offstar` is to compute this address into a register or to leave the subtree in a state where it can be readily transformed into an OREG node.

12.4 The "set" procedures

The next five procedures, with names beginning with `set`, represent sections of code which, if they were not machine-dependent, would occur in-line in the procedure `order`. As will be recalled, the general strategy of `order`, which these procedures follow, is to perturb the tree and try again. They are coded as a sequence of actions in order of increasing severity, or desperation. Each `return` statement can be read as "have another try to match a template".

12.4.1 `setincr` *(3378)* is called by `order` at line 1713 to perform any machine-dependent processing of INCR or DECR nodes before `order` resumes its normal procedures. The PDP11 does not seem to offer any interesting possibilities here.

12.4.2 `setstr` *(3383)* is called by `order` at line 1732 to sort out structure assignments and this is considered to be an entirely machine-dependent affair. If `setstr` cannot find some way to perturb the current set-up, then there is no machine-independent recipe to fall back on.

3384: Get the value from the right-hand subtree into a temporary register.

3388: Look down the left subtree. If the node is not a NAME or an OREG, then it had better be a UNARY MUL, whose subtree can be computed into a temporary register.

This procedure offers two apparently different ways to fail: the call on `cerror` at line 3390, or the `return` at line 3394. However the latter will lead very rapidly to the call on `cerror` at line 1604.

12.4.3 `setasop` *(3398)* is called by `order` at line 1736 to provide machine-dependent tree rewriting for assignment operators. As with the rest of its sister procedures, and the related code in `order`, the basic idea is to keep stirring, to keep chopping bits off the tree (the recursive calls to `order`), and then trying again until it is possible to generate code.

In this case, there is also the chance to rewrite the tree in a major way, so as to separate the actions of assignment and the basic arithmetic operations. The various alternatives are arranged

```
	case ASG ER:
		if( sul == 0 || p->left->op == REG ){
			if( p->left->type == CHAR ||
			p->left->type == UCHAR )
				goto rew;  /* rewrite */
			order( p->right, INAREG|INBREG );
			return( 1 );
			}
		goto leftadr;
		}

	if( sur == 0 ){

	leftadr:
		/* easy case: if addressable, do left value, op, store */
		if( sul == 0 ) goto rew;  /* rewrite */

		/* harder: make aleft address, val, op, and store */
		if( p->left->op == UNARY MUL ){
			offstar( p->left->left );
			return( 1 );
			}
		if(p->left->op == FLD && p->left->left->op == UNARY MUL){
			offstar( p->left->left->left );
			return( 1 );
			}
	rew:	/* rewrite accounting for autoincrement, autodecrement */
		q = p->left;
		if( q->op == FLD ) q = q->left;
		if( q->op != UNARY MUL || shumul(q->left) != STARREG )
			return(0); /* let reader.c do it */

		/* mimic code from reader.c */
		p2 = tcopy( p );
		p->op = ASSIGN;
		reclaim( p->right, RNULL, 0 );
		p->right = p2;

		/* now, zap INCR on right, ASG MINUS on left */
		if( q->left->op == INCR ){
			q = p2->left;
			if( q->op == FLD ) q = q->left;
			if( q->left->op != INCR )
				cerror( "bad incr rewrite" );
			}
		else if( q->left->op != ASG MINUS )
			cerror( " bad -= rewrite" );

		q->left->right->op = FREE;
		q->left->op = FREE;
		q->left = q->left->left;

		/* now, resume reader.c rewriting code */
		canon(p);
		rallo( p, p->rall );
		order( p2->left, INTBREG|INTAREG );
		order( p2, INTBREG|INTAREG );
		return( 1 );
		}

	/* harder case: do right, left address, left value, op, store */
	if( p->right->op == UNARY MUL ){
		offstar( p->right->left );
		return( 1 );
		}
	/* sur> 0, since otherwise, done above */
	/* make lhs addressable */
	if( p->right->op == REG ) goto leftadr;
	order( p->right, INAREG|INBREG );
	return( 1 );
	}
```

in order of increasing complexity, as determined by the SU numbers.

3411: If the operand type is not a single word, don't attempt anything special yet.

3413: If the right subtree operand type is "character", then get the right operand into a register and try for a template match.

3419: If the left subtree is easy, get the result of the right subtree into a register. Templates for `ASG ER` begin at line 5231. Note that the PDP11 `xor` instruction is unusual in that it expects the source (i.e., the right operand) to be in a register.

3430: The code from here to line 3477 accounts for all possibilities for which the right subtree represents a readily accessible operand.

3437: Try to get the left subtree into a form where it can be converted into an `OREG`.

3441: Get an assignment into a field into a form that can be matched by a template. The templates for this purpose begin at line 4782. They are associated with the `ASSIGN` operator and require the left subtree to have the "shape" `SFLD`.

3448: The code to handle the case where the left operand is not directly addressable begins at line 1740.

3451: The code from here to line 3475 is a modification of the code on lines 1740 through 1750. The important difference is the handling of `INCR` and `ASG MINUS` operations as the side effects of autoincrement and autodecrement addressing.

3480: The alternative situation, where `sur`, the number of registers needed in the computation of the right subtree, is non-zero, begins here. Try to simplify the right subtree.

3487: Get the right operand into a register.

12.4.4 `setasg` *(3492)* is called by `order` at line 1754 to rewrite the tree in order to handle structure assignments.

3495: Start by simplifying the right subtree.

3502: If the right operand is not in a register, and the operand type is `FLOAT` or `DOUBLE`, then get it into a register.

3507: It would seem simpler to use `shumul` directly here.

3512: Anyway, get the left subtree into a state where it can be made into an `OREG`.

3517: At this point, several attempts at a template match have been made without success. As a last resort, force the right operand into a register.

12.4.5 `setbin` *(3525)* is called by `order` at line 1759 to rewrite a tree whose root node is a binary operator. The pattern and style of this procedure are similar to those found in `setasop` (3398) and `setasg`, which have just been examined*. A series of stratagems are provided that can be invoked one by one until a match is achieved.

* The present author finds a number of features of the program strategy at this point to be somewhat unsatisfying. There are the many points of interaction between the contents of `table`, the strategy of `sucomp`, and that of the `set` procedures. This tripartite arrangement is, for the uninitiated, almost unfathomable. Then there is, for example, the use of `offstar` to modify a subtree, so that subsequently `oreg2` (1988) can can convert it into an OREG node. Surely some more direct means to achieve the same end would be possible.

```
/* ---------------------------------------------------- */

setasg( p ) register NODE *p; {
	/* setup for assignment operator */

	if( p->right->su != 0 && p->right->op != REG ) {
		if( p->right->op == UNARY MUL )
			offstar( p->right->left );
		else
			order( p->right, INAREG|INBREG|SOREG|SNAME|SCON );
		return(1);
		}
	if( p->right->op != REG &&
			( p->type == FLOAT || p->type == DOUBLE ) ) {
		order( p->right, INBREG );
		return(1);
		}
	if( p->left->op == UNARY MUL &&
			!tshape( p->left, STARREG|STARNM ) ){
		offstar( p->left->left );
		return(1);
		}
	if( p->left->op == FLD && p->left->left->op == UNARY MUL ){
		offstar( p->left->left->left );
		return(1);
		}
	/* if things are really strange, get rhs into a register */
	if( p->right->op != REG ){
		order( p->right, INAREG|INBREG );
		return( 1 );
		}
	return(0);
	}
/* ---------------------------------------------------- */

setbin( p ) register NODE *p; {
	register NODE *r, *l;

	r = p->right;
	l = p->left;

	if( p->right->su == 0 ){ /* rhs is addressable */
		if( logop( p->op ) ){
			if( l->op == UNARY MUL && l->type != FLOAT &&
					shumul( l->left ) != STARREG )
				offstar( l->left );
			else order(l,INAREG|INTAREG|INBREG|INTBREG|INTEMP);
			return( 1 );
			}
		if( !istnode( l ) ){
			order( l, INTAREG|INTBREG );
			return( 1 );
			}
		/* rewrite */
		return( 0 );
		}
	/* now rhs is complicated: must do both sides into registers */
	/* do the harder side first */

	if( logop( p->op ) ){
		/* relational: do both sides into regs if need be */

		if( r->su > l->su ){
			if( niceuty(r) ){
				offstar( r->left );
				return( 1 );
				}
			else if( !istnode( r ) ){
				order(r,INTAREG|INAREG|INTBREG|INBREG|INTEMP);
				return( 1 );
				}
			}
```

12.5 `niceuty` (3604)

The name of this procedure, which is a PDP11 exclusive, seems to be a contraction for "nice unary type": It is called by `setbin` at lines 3553, 3562, 3566 and 3584. If the result returned is true, a call to `offstar` follows.

3607: The entire procedure is a single `return` statement, which returns true if the node is a UNARY MUL, the operand type is not exotic, and `shumul` finds it acceptable. The question to be answered is whether the subtree should be turned into a direct address or OREG. However this is not to be done if the operand is already directly addressable (e.g., if the shape of the tree is STARREG).

(`shumul` (4147) can return three possible values: STARREG, which is not acceptable here, STARNM, which is, and 0, which most probably should not be acceptable.)

12.6 `notoff` (3613)

This procedure, which is called by `oreg2` at line 2065, is asked to inspect the size of the offset at an OREG and to pronounce upon its suitability. For the PDP11 and the VAX11/780, and most other machines, there is no problem, but, for machines in the class of the IBM 360/370, offsets must be restricted to 12 bit positive integers.

12.7 `genargs` (3623)

This procedure is called by `gencall` (4032), which is itself called by `order` at line 1688. It generates code to assemble the arguments in the object time stack. Since there is a convention in C that procedure arguments should be addressable as the elements of an array, and since stacks grow downwards on the PDP11, this implies processing the arguments from right to left*.

3628: Link through the argument list recursively to get to the last argument. (To reverse the order in which the arguments are evaluated, it suffices to invert the references to "right" and "left" on lines 3629 and 3631.)

3633: If the argument is a structure, copy the structure into the stack. This is a special case since the method for copying the structure onto the stack can vary, depending on the type of stack architecture (whether it is maintained via hardware or software) and the direction of stack growth.

3657: The "cookie" passed to `expand` is only meaningful when the character string contains an `'F'`. The string "AR" will be found to result in a call to `adrput` with `getlr( p, 'R' )` as argument. The string "`Z-`" results in the instruction address "`-(sp)`".

3664: All other arguments get placed on the stack through a call to `order` with the "cookie" FORARG. Templates for the "cookie" may be found at lines 4849, 4885, 4891 and 4897. Note that each of these leaves a value in the stack.

12.8 `argsize` (3668)

`gencall` calls `argsize` to determine, in advance, the number of locations that will be occupied by the arguments in the stack. `gencall` subsequently passes this value to `popargs` to generate code that will cut the stack back after the called procedure returns.

Two questions arise: why can't `genargs` produce this value as a side effect? and why does `argsize` search the argument list in a different order from `genargs`?*

---

* For machines such as the IBM/370, where the stack grows in the positive direction, the arguments must be processed from left to right.

* The second question is readily answered. The two procedures used to be the same, but the PDP11 version of `genargs` was changed to generate the arguments in the reverse order.

```
		if( niceuty(l) ){
			offstar( l->left );
			return( 1 );
			}
		else if( niceuty(r) ){
			offstar( r->left );
			return( 1 );
			}
		else if( !istnode( l ) ){
			order( l, INTAREG|INAREG|INTBREG|INBREG|INTEMP );
			return( 1 );
			}
		if( !istnode( r ) ){
			order( r, INTAREG|INAREG|INTBREG|INBREG|INTEMP );
			return( 1 );
			}
		cerror( "setbin can't deal with %s", opst[p->op] );
		}

	/* ordinary operator */
	if( !istnode(r) && r->su > l->su ){
		/* if there is a chance to make it addressable, try... */
		if( niceuty(r) ){
			offstar( r->left );
			/* hopefully, it is addressable by now */
			return( 1 );
			}
		/* anything goes on rhs */
		order( r, INTAREG|INAREG|INTBREG|INBREG|INTEMP );
		return( 1 );
		}
	else {
		if( !istnode( l ) ){
			order( l, INTAREG|INTBREG );
			return( 1 );
			}
		/* rewrite */
		return( 0 );
		}
	}
/* ----------------------------------------------------- */

niceuty( p ) register NODE *p; {
	register TWORD t;

	return( p->op == UNARY MUL && (t=p->type)!=CHAR &&
		t!= UCHAR && t!= FLOAT &&
		shumul( p->left) != STARREG );
	}
/* ----------------------------------------------------- */

notoff( t, r, off, cp) TWORD t; CONSZ off; char *cp; {
	/* is it legal to make an OREG or NAME entry which has an
	/* offset of off, (from a register of r), if the
	/* resulting thing had type t */

	/* return( 1 ); /* NO */
	return(0); /* YES */
	}
/* ===================================================== */
```

Chapter 13: The File "local2.c" Part One

Like `order.c`, this file also contains procedures that are machine-dependent and have widely diverse functions. For the most part, these procedures are simply sequences of code which have been quarantined away from the machine-independent portions of the compiler. Only a few of these, notably `cbgen`, `szty` and `shltype`, are called from more than one place in the machine-independent code.

The first group of procedures in the file are connected with the procedure `main` (0961):

1. `setregs` sorts out the temporary registers.
2. `eobl2` does end of block processing.
3. `lineid` identifies the current source line.
4. `hardops` converts some operators to calls on library routines.
5. `optim2` rewrites `(ASG) AND` nodes.
6. `myreader` invokes `hardops` and `optim2`.

The next procedure is `cbgen`, which is concerned with the generation of assembly language branch instructions. Since comparisons of long variables on the PDP11 involve double words, this is not a completely trivial procedure.

The second major group of procedures are associated with code for procedure calls:

1. `callreg` specifies the register in which values are to be returned.
2. `genscall` handles calls for procedures that return structures.
3. `gencall` generates the normal procedure call sequence.
4. `popargs` generates code for cutting the stack back.

The remaining procedures of the file are the subject of Chapter Fourteen.

13.1 Declarations

The definition of `BITMASK` (3697) provides a set of masks with `(SZINT - n)` significant zeroes and `n` ones.

The pointer `brnode` and the integer variable `brcase` are used by `zzzcode` to transmit information indirectly to `cbgen`.

`rnames` provides a set character strings for both diagnostic and code generation purposes.

`rstatus` has an entry for each of the processor registers (fourteen in all on the PDP11). Each entry defines whether the register is of type A or B, and whether it may be used as a temporary scratch register. The status of type A registers may change from temporary to non-temporary, or vice versa, at the beginning of each block, when `setregs` (3739) is called. The initial content of `rstatus` is used by `allo0` (2458) in determining values for `maxa`, `mina`, `maxb` and `minb`.

`respref` is an array which provides directives to `reclaim` at line 2740 for selecting the best alternative if the result of a calculation is available in more than one form. The structure `respref` is declared at line 0524 and consists of two integer elements, `cform` and `mform`. On the face of it, this array does not seem to be very machine-dependent. It would be easier,

```
genargs( p) register NODE *p; {
	/* generate code for the arguments */
	register size;

	/*  first, do the arguments on the right (last->first) */
	while( p->op == CM ){
		genargs( p->right );
		p->op = FREE;
		p = p->left;
		}
	if( p->op == STARG ){ /* structure valued argument */

		size = p->stsize;
		if( p->left->op == ICON ){
			/* make into a name node */
			p->op = FREE;
			p= p->left;
			p->op = NAME;
			}
		else {
			/* make it look beautiful... */
			p->op = UNARY MUL;
			canon( p );  /* turn it into an oreg */
			if( p->op != OREG ){
				offstar( p->left );
				canon( p );
				if( p->op != OREG ) cerror( "stuck starg" );
				}
			}

		p->lval += size;  /* end of structure */
		/* put on stack backwards */
		for( ; size>0; size -= 2 ){
			p->lval -= 2;
			expand( p, RNOP, "	mov	AR,Z-\n" );
			}
		reclaim( p, RNULL, 0 );
		return;
		}
	/* ordinary case */

	order( p, FORARG );
	}
/* ------------------------------------------------- */

argsize( p ) register NODE *p; {
	register t;
	t = 0;
	if( p->op == CM ){
		t = argsize( p->left );
		p = p->right;
		}
	if( p->type == DOUBLE || p->type == FLOAT ){
		SETOFF( t, 2 );
		return( t+8 );
		}
	else if( p->type == LONG || p->type == ULONG ) {
		SETOFF( t, 2);
		return( t+4 );
		}
	else if( p->op == STARG ){
		SETOFF( t, p->stalign );  /* alignment */
		return( t + p->stsize );  /* size */
		}
	else {
		SETOFF( t, 2 );
		return( t+2 );
		}
	}

/* ------------------------------------------------- */
```

at least for the reader, if `respref` dealt with only one "original cookie" at a time, and for each of these, listed the acceptable alternatives in order of decreasing attractiveness.

13.2 `setregs` (3739)

`setregs` is called by `main` at line 1007, during the initialization phase at the beginning of each block. (This procedure is actually simpler than a first glance at the code suggests.)

3743: `maxtreg` is the number of the last type A register assigned as a register variable. (These are allocated in descending order.) Set `fregs`, which specifies the number of temporary registers, to one greater than `maxtreg`, except that it must be at least `MINRVAR` (defined to have the value two).

3744: Use the "x" debugging flag to further limit the value of `fregs`. Useful for debugging register allocation strategies.

3749: Make sure that `fregs` is not too large. (This is really a check on `maxtregs`.)

3750: Adjust the status of all the type A registers, which may sometimes be used as temporary registers and sometimes not. (Remember, this is done at the beginning of each block.)

13.3 `eobl2` (3755)

This procedure is also called by `main`, at line 1012, after each block has been processed, to perform "end of block" chores.

3758: Determine the maximum growth of the temporary storage section of the stack. This value has to be discounted for the "automatic" growth due to the normal procedure prologue (the procedure `csv`). For the PDP11, `csv` unconditionally stores `R4`, `R3` and `R2` in the stack. This is three words, or 48 bits (the value of `AUTOINIT`).

3762: Pass the stack growth value to the assembler via a constant definition. This value is used by the assembler to replace the symbolic name in an instruction of the type

```
sub $.Fn,sp
```

which is used to advance the stack pointer at procedure entry time.

3763: If any floating point operations have been generated, define the global symbol `fltused` to the assembler. This is a flag to the loader that it should load the "floating point" versions of certain library routines, especially `printf`. This action is really needed only once per program, not for every block. An alternative would be to replace the expression tested at line 3763 by `(fltused > 0)`, and to replace line 3764 by

```
fltused -= 10000;
```

13.4 `lineid` (3770)

`lineid` is called by `main`, at line 1022, to place a comment in the assembler listing to identify the origin (source file, line number) of the expression evaluated by the code that follows.

13.5 `where` (3776)

The procedure provided here is a dummy. It is referenced in each of the three "error" procedures, `cerror` (0621), `uerror` (0599) and `werror` (0612), with the intention, presumably, that it should provide some indication in terms of a reference into the source code, as to where the trouble being reported occurred.

```
# include "mfile2"
/* a lot of the machine dependent parts of the second pass */

# define BITMASK(n) ((1L<<n)-1)

NODE *brnode;
int brcase;

int toff = 0; /* number of stack locations used for args */
/* ------------------------------------------------- */

char *
rnames[]= {  /* keyed to register number tokens */

	"r0", "r1",
	"r2", "r3", "r4",
	"r5", "sp", "pc",

	"fr0", "fr1", "fr2", "fr3",
	"fr4", "fr5",	/* not accumulators - used for temps */
	};
/* ------------------------------------------------- */

int rstatus[] = {
	SAREG|STAREG, SAREG|STAREG,
	SAREG|STAREG, SAREG|STAREG,
	SAREG|STAREG,	/* use as scratch if not reg var */
	SAREG, SAREG, SAREG,

	SBREG|STBREG, SBREG|STBREG, SBREG|STBREG, SBREG|STBREG.
	SBREG, SBREG,
	};
/* ------------------------------------------------- */

struct respref
respref[] = {
	INTAREG|INTBREG,  INTAREG|INTBREG,
	INAREG|INBREG,
		INAREG|INBREG|SOREG|STARREG|SNAME|STARNM|SCON,
	INTEMP,     INTEMP,
	FORARG,     FORARG,
	INTAREG,    SOREG|SNAME,
	0,    0 };
/* ------------------------------------------------- */

setregs(){ /* set up temporary registers */
	register i;

	/* use any unused variable registers as scratch registers */
	fregs = maxtreg>=MINRVAR ? maxtreg + 1 : MINRVAR;
	if( xdebug ){
		/* -x changes number of free regs to 2, -xx to 3, etc */
		if( (xdebug+1) < fregs ) fregs = xdebug+1;
		}
	/* NOTE: for pdp11 fregs <= 4 for float regs */
	if( fregs > 4 ) fregs = 4;
	for( i=MINRVAR; i<=MAXRVAR; i++ )
		rstatus[i] = i<fregs ? SAREG|STAREG : SAREG;
	}
/* ------------------------------------------------- */

eobl2(){
	OFFSZ spoff;	/* offset from stack pointer */

	spoff = maxoff;
	if( spoff >= AUTOINIT ) spoff -= AUTOINIT;
	spoff /= SZCHAR;
	SETOFF(spoff,2);
	printf( "   .F%d = %Ld.\n", ftnno, spoff );
	if( fltused ) {
		fltused = 0;
```

```
			printf( "	.globl	fltused\n" );
			}
		}
/* -------------------------------------------------- */

lineid( l, fn ) char *fn; {
		/* identify line l and file fn */
		printf( "/	line %d, file %s\n", l, fn );
		}
/* -------------------------------------------------- */

where ( c ) char c; {
		/* VOID */
		}
/* ================================================== */

struct functbl {
		int fop;
		TWORD ftype;
		char *func;
		} opfunc[] = {

		MUL,		LONG,		"lmul",
		DIV,		LONG,		"ldiv",
		MOD,		LONG,		"lrem",
		ASG MUL,	LONG,		"almul",
		ASG DIV,	LONG,		"aldiv",
		ASG MOD,	LONG,		"alrem",
		MUL,		ULONG,		"lmul",
		DIV,		ULONG,		"uldiv",
		MOD,		ULONG,		"ulrem",
		ASG MUL,	ULONG,		"almul",
		ASG DIV,	ULONG,		"auldiv",
		ASG MOD,	ULONG,		"aulrem",
		0,	0,	0 };
/* -------------------------------------------------- */

hardops(p)  register NODE *p; {
		/* change hard to do operators into function calls.
		   for pdp11 do long * / %	*/
		register NODE *q;
		register struct functbl *f;
		register o;
		register TWORD t;

		o = p->op;
		t = p->type;
		if( t!=LONG && t!=ULONG ) return;

		for( f=opfunc; f->fop; f++ ) {
			if( o==f->fop && t==f->ftype ) goto convert;
			}
		return;

		/* need address of left node for ASG OP */
		/* WARNING - this won't work for long in a REG */
		convert:
		if( asgop( o ) ) {
			switch( p->left->op ) {

			case UNARY MUL:	/* convert to address */
				p->left->op = FREE;
				p->left = p->left->left;
				break;

			case NAME:	/* convert to ICON pointer */
				p->left->op = ICON;
				p->left->type = INCREF( p->left->type );
				break;

			case OREG:	/* convert OREG to address */
				p->left->op = REG;
```

```
                        p->left->type = INCREF( p->left->type );
                        if( p->left->lval != 0 ) {
                              q = talloc();
                              q->op = PLUS;
                              q->rall = NOPREF;
                              q->type = p->left->type;
                              q->left = p->left;
                              q->right = talloc();

                              q->right->op = ICON;
                              q->right->rall = NOPREF;
                              q->right->type = INT;
                              q->right->name[0] = '\0';
                              q->right->lval = p->left->lval;
                              q->right->rval = 0;

                              p->left->lval = 0;
                              p->left = q;
                              }
                        break;

                  default:
                        cerror( "Bad address for hard ops" );
                        /* NO RETURN */

                        }
                  }

            /* build comma op for args to function */
            q = talloc();
            q->op = CM;
            q->rall = NOPREF;
            q->type = INT;
            q->left = p->left;
            q->right = p->right;
            p->op = CALL;
            p->right = q;

            /* put function name in left node of call */
            p->left = q = talloc();
            q->op = ICON;
            q->rall = NOPREF;
            q->type = INCREF( FTN + p->type );
            strcpy( q->name, f->func );
            q->lval = 0;
            q->rval = 0;

            return;
            }
      /* ---------------------------------------------------- */

      optim2( p ) register NODE *p; {
            /* do local tree transformations and optimizations */

            register NODE *r;

            switch( p->op ) {

            case AND:
            /* commute L and R to eliminate complements and constants */
                  if( p->left->op==ICON || p->left->op==COMPL ) {
                        r = p->left;
                        p->left = p->right;
                        p->right = r;
                        }
            case ASG AND:
                  /* change meaning of AND to ~R&L - bic on pdp11 */
                  r = p->right;
                  if( r->op==ICON ) { /* complement constant */
                        r->lval = ~r->lval;
                        }
```

13.6 hardops (3802)

This procedure is passed by the procedure myreader (3926) as the procedure argument to walkf (0688). (myreader is called at line 1031 under the alias of MYREADER.) The intention is to perform a preorder walk of the expression tree looking for certain combinations of operator/operand type for which the code generated will be calls on standard library subroutines. The list of such combinations for the PDP11 can be found starting at line 3787. hardops is called before canon, which calls oreg2 (1988), so that OREG nodes will not have to be unraveled.

3812: The only operand types of interest are LONG or ULONG.

3814: Locate the appropriate entry in opfunc (3785), or return if none exists.

3822: If this is an assignment operator, the value presented to the library routine for the left subtree must be an address, so look at the root of the left subtree.

3825: The root is a UNARY MUL. Just throw it away and find the address.

3830: NAME nodes can become address constants.

3835: In spite of the comment above, there may still be OREG nodes that need to be expanded back into explicit arithmetic expressions. (Although the first pass of the Portable C compiler does not generate OREG nodes, the first pass of the Fortran 77 compiler may do so!)

3865: Build a subtree representing a function call, with an argument list, and the name of the appropriate function.

13.7 optim2 (3888)

Like hardops just described, this procedure is passed by myreader (3926) to walkf as its procedure argument. This results in a preorder traversal of the tree, with optim2, like hardops before it, applied at each node.

The task of optim2 is to rewrite the tree for AND and ASG AND to reflect the properties of the PDP11's bic instruction. This is an asymmetric operation, which, in the absence of a better alternative, is used to implement the symmetric AND operation. The bic instruction computes ~R&L i.e. the conjunction of the "destination" operand with the complement of the "source" operand (where "destination" and "source" are used in the same sense as in the PDP11 Processor Handbook.)

3898: Interchange the left and right subtrees if the left subtree is a constant, or begins with a COMPL operation.

3904: For both ASG AND and AND operators, ...

3905: complement the right hand subtree, which will be easy if it is constant, ...

3908: only a little harder if the right hand subtree has a COMPL operator at its root (this is a unary operator, for which the corresponding node is simply thrown away, since two COMPLs cancel each other).

3912: The remaining case requires the addition of a new node to the tree, to represent the COMPL operation which must be inserted.

```
			else if( r->op==COMPL ) { /* ~~A => A */
				r->op = FREE;
				p->right = r->left;
				}
			else { /* insert complement node */
				p->right = talloc();
				p->right->op = COMPL;
				p->right->rall = NOPREF;
				p->right->type = r->type;
				p->right->left = r;
				p->right->right = NULL;
				}
			break;

			}
		}
/* -------------------------------------------------- */

myreader(p) register NODE *p; {
	walkf( p, hardops );	/* convert ops to function calls */
	canon( p );		/* expands r-vals for fileds */
	walkf( p, optim2 );
	toff = 0;  /* stack offset swindle */
	}
/* ================================================== */

char *
ccbranches[] = {
	"	jeq	L%d\n",
	"	jne	L%d\n",
	"	jle	L%d\n",
	"	jlt	L%d\n",
	"	jge	L%d\n",
	"	jgt	L%d\n",
	"	jlos	L%d\n",
	"	jlo	L%d\n",
	"	jhis	L%d\n",
	"	jhi	L%d\n",
	};
/* -------------------------------------------------- */

/*	long branch table

   This table, when indexed by a logical operator,
   selects a set of three logical conditions required
   to generate long comparisons and branches.  A zero
   entry indicates that no branch is required.
   E.G.:  The <= operator would generate:
	cmp	AL,AR
	jlt	lable / 1st entry LT -> lable
	jgt	1f    / 2nd entry GT -> 1f
	cmp	UL,UR
	jlos	lable / 3rd entry ULE -> lable
   1:
 */

int lbranches[][3] = {
	/*EQ*/		0,	NE,	EQ,
	/*NE*/		NE,	0,	NE,
	/*LE*/		LT,	GT,	ULE,
	/*LT*/		LT,	GT,	ULT,
	/*GE*/		GT,	LT,	UGE,
	/*GT*/		GT,	LT,	UGT,
	/*ULE*/		ULT,	UGT,	ULE,
	/*ULT*/		ULT,	UGT,	ULT,
	/*UGE*/		UGT,	ULT,	UGE,
	/*UGT*/		UGT,	ULT,	UGT,
	};
/* -------------------------------------------------- */

/* logical relations when compared in reverse order (cmp R,L) */
short revrel[]={EQ, NE, GE, GT, LE, LT, UGE, UGT, ULE, ULT};
```

13.8 myreader (3926)

With hardops and optim2 already discussed, the function of myreader is now fairly clear. Under the name MYREADER (defined at line 0348) it is invoked by main at line 1031 for each expression tree, after the latter has been read in, but before it is passed to delay. myreader performs various "one time" changes to the tree (c.f. canon, which may be called many times).

3928: canon is called after the call on hardops, so that the latter will not have the problem, already alluded to, of unraveling OREG nodes which canon may ravel. On the other hand, canon is called before optim2 because the rewriting of field extractions may introduce additional AND nodes that optim2 must attend to.

3930: The comment here begs a further comment*.

13.9 cbgen (3981)

This procedure is called in several places from cbranch (1806), order (1524) and zzzcode (4415). The first parameter is, for the most part, simply zero. It may also be o (see lines 1852 and 3981)! At lines 1852, 1915 and 1916, it is clearly a relational operator, e.g. EQ. Only at line 4436 is the value hard to predict, since the value is then taken from one of the table entries.

The third parameter is normally 'I' (for "integer"), but it may also be 'F' (for "floating point") in the call from line 4436.

3986: This routine envisages three main possibilities. The first is that o is 0, so that the branch to be generated is unconditional, and rapidly disposed of.

3987: The second possibility is that the first argument is in error.

3989: The third possibility is much more complicated, and derives from the interaction between operator templates in table and the machinations derived therefrom by zzzcode.

3990: brcase is used to transfer information from zzzcode to cbgen when the comparisons involve long comparisons (see lines 4883, 4956).

3994: Comparison with longs involve two stages of testing. (See the comment which begins at line 3951.) The next few lines are brutal, but straightforward enough.

4010: If it is not a "long" comparison, and if the mode is 'F', use the array revrel (3979) to reverse the sense of the comparison.

4011: ccbranches is declared at line 3935.

4015: Reset brcase and brnode before exiting so that the default case will be used next time if zzzcode has not prepared a long comparison.

13.10 callreg (4021)

This procedure is called once, by freereg at line 2555. It reaffirms the convention that the results from procedure calls are returned via R0 or FR0, as appropriate.

---

* Lee Benoy notes "toff should have been reset to zero in the previous expression evaluation. This is just to be doubly certain."

```
cbgen( o, lab, mode ) {
	/*   printf conditional and unconditional branches */
	register *plb;
	int lab1f;

	if( o == 0 ) printf( "   jbr   L%d\n", lab );
	else  if( o > UGT ) cerror( "bad conditional branch: %s",
			opst[o] );
	else {
		switch( brcase ) {

		case 'A':
		case 'C':
			plb = lbranches[ o-EQ ];
			lab1f = getlab();
			expand( brnode, FORCC, brcase=='C' ?
				"\tcmp\tAL,AR\n" : "\ttst\tAR\n" );
			if( *plb != 0 )
				printf( ccbranches[*plb-EQ], lab);
			if( *++plb != 0 )
				printf( ccbranches[*plb-EQ], lab1f);
			expand( brnode, FORCC, brcase=='C' ?
				"\tcmp\tUL,UR\n" : "\ttst\tUR\n" );
			printf( ccbranches[*++plb-EQ], lab);
			deflab( lab1f );
			reclaim( brnode, RNULL, 0 );
			break;

		default:
			if( mode=='F' ) o = revrel[ o-EQ ];
			printf( ccbranches[o-EQ], lab );
			break;
			}

		brcase = 0;
		brnode = 0;
		}
	}
/* --------------------------------------------------- */

callreg(p) NODE *p; {
	return( (p->type==DOUBLE||p->type==FLOAT) ? FR0 : R0 );
	}
/* --------------------------------------------------- */

genscall( p, cookie ) register NODE *p; {
	/* structure valued call */
	return( gencall( p, cookie ) );
	}
/* --------------------------------------------------- */

gencall( p, cookie ) register NODE *p; {
	/* generate the call given by p */
	register temp;
	register m;

	if( p->right ) temp = argsize( p->right );
	else temp = 0;

	if( p->right ){ /* generate args */
		genargs( p->right );
		}

	if( !shltype( p->left->op, p->left ) ) {
		order( p->left, INAREG|SOREG );
		}

	p->op = UNARY CALL;
	m = match( p, INTAREG|INTBREG );
	popargs( temp );
	return(m != MDONE);
	}
/* --------------------------------------------------- */
```

13.11 `genscall` (4026)

This procedure simply dummies up a call to `gencall`. It is to be compared with `genfcall`, which is *defined* as `gencall` at line 0341. (It would seem preferable if both this procedure and its predecessor, `callreg`, were made into `#define` statements.)

13.12 `gencall` (4032)

`gencall` is called directly by `order` at line 1688, and via its aliases, `genfcall` and `genscall`, at lines 1681 and 1695 respectively. Any distinctions between these that may be drawn on some machines are not visible with the PDP11.

4037: Determine how far the stack will grow at run-time when the arguments are generated.

4040: If there are arguments, generate the code that will bring them into the stack.

4044: If the left subtree, which must reduce to the address of a function, is not in a state to be passed to the subroutine call instruction, get its result into a type A register or, at least, transform the subtree into the shape of an OREG.

4048: With the arguments in the stack, convert the call to a UNARY CALL.

4049: Now call `match` to match the UNARY CALL, with the result, if any, going into a temporary register (as determined by `callreg`).

4050: Call `popargs` to generate an instruction which will cut the stack back by the amount calculated by `argsize`.

4051: Return a non-zero result if `match` did not return MDONE. (Back in `order`, this will result in a transfer to `nomat` at line 1603, and a call to `cerror`.)

13.13 `popargs` (4055)

This procedure is called only once, by `gencall` at line 4050. At least in the PDP11 version of the Portable C compiler, there is no real reason for its separate existence.

4058: `toff` keeps track of the size of all arguments in the stack when procedure calls are nested. `size` (i.e. the variable `temp` declared by `gencall` (4032)) accounts only for the arguments of the current procedure, and does so in units of bytes. Note that if there have been no nested calls and there was only one single-word argument, i.e., `toff==1`, `size==2`, then no stack adjustment is required.

For arguments to procedure calls, the convention is that the stack pointer will be pointing initially at the location in the stack where the procedure result may be stored. This same location may also be used for the first argument to the procedure. (See also lines 4543 to 4550.) Since procedures return their result via R0 or FR0, storage of the result in the stack is only important for the case of nested procedure calls, when the "cookie" FORARG implies moving the value of R0 or FR0 into the stack.

4060: Generate the most efficient instruction to increment the stack pointer by the required amount. (On the PDP11, the stack pointer is incremented to cut back the stack, because stacks grow in the negative direction.)

```

popargs( size ) register size; {
	/* pop arguments from stack */

	toff -= size/2;
	if( toff == 0 && size >= 2 ) size -= 2;
	switch( size ) {
	case 0:
		break;
	case 2:
		printf( "	tst	(sp)+\n" );
		break;
	case 4:
		printf( "	cmp	(sp)+,(sp)+\n" );
		break;
	default:
		printf( "	add	$%d.,sp\n", size);
		}
	}
/* ================================================== */

nextcook( p, cookie ) NODE *p; {
	/* we have failed to match p with cookie; try another */
	if( cookie == FORREW ) return( 0 );  /* hopeless! */
	if( !(cookie&(INTAREG|INTBREG)) ) return( INTAREG|INTBREG );
	if( !(cookie&INTEMP) && asgop(p->op) )
		return( INTEMP|INAREG|INTAREG|INTBREG|INBREG );
	return( FORREW );
	}
/* -------------------------------------------------- */

lastchance( p, cook ) NODE *p; {
	/* forget it! */
	return(0);
	}
/* -------------------------------------------------- */

rewfld( p ) NODE *p; {
	return(1);
	}
/* -------------------------------------------------- */

spsz( t, v ) TWORD t; CONSZ v; {

	/* is v the size to increment something of type t */

	if( !ISPTR(t) ) return( 0 );
	t = DECREF(t);

	if( ISPTR(t) ) return( v == 2 );

	switch( t ){

	case UCHAR:
	case CHAR:
		return( v == 1 );

	case INT:
	case UNSIGNED:
		return( v == 2 );

	case FLOAT:
		return( v == 4 );

	case DOUBLE:
		return( v == 8 );
		}

	return( 0 );
	}
/* -------------------------------------------------- */
```

The chapter discusses the remaining procedures in the file `local2.c`. The first group consists of three easy ones:

1. `nextcook` provides an alternative goal when the original one seems unattainable.
2. `lastchance` is a desperation move that is ignored on the PDP11.
3. `rewfld` is a chance to invoke limited hardware resources for field extraction.

The next set of procedures is used for the evaluation of types and shapes:

1. `spsz` checks whether hardware autoincrement or decrement will work correctly.
2. `szty` determines the number of registers to store a given type.
3. `shltype` determines whether a particular subtree has the shape of a leaf.
4. `shumul` determines the shape of a tree whose root is `UNARY MUL`.
5. `special` looks for machine-dependent shapes that may receive special treatment.
6. `shtemp` determines whether a particular subtree has the shape of temporary storage.
7. `flshape` determines whether a subtree is ready for field extraction.

The last group of procedures is associated with the expansion of strings, taken from templates in `table`, into assembly language statements:

1. `acon` emits a constant value as part of an address.
2. `adrcon` emits a special kind of constant.
3. `adrput` emits the address of a source or destination.
4. `conput` emits a non-address constant.
5. `insput` is not used for the PDP11.
6. `upput` does the half of long variables that is not handled by `adrput`.
7. `rmove` generates a register-to-register move.
8. `hopcode` emits an operator mnemonic selected from a table.
9. `zzzcode` does extra, machine-dependent things for `expand`.

14.1 `nextcook` (4075)

`nextcook` is called by `order` at line 1563 when an initial attempt to generate code for the subtree has ended in failure. If the subtree is to generate an intermediate result, it is possible that `order` may still be able to succeed, if the conditions as to where the result may appear are relaxed somewhat.

`nextcook` returns 0, meaning "hopeless", in a situation where code cannot be generated. (This is likely to be a common occurrence in the early days of a new version of the compiler.) `nextcook` may return FORREW, if the only possibility lies in re-organizing the tree in some way. (This is the only alternative if the original goal was FOREFF, or FORARG or FORCC.)

```

szty(t) TWORD t; { /* size, in words, needed for thing of type t */
	/* really is the number of registers to hold type t */
	switch( t ) {

	case LONG:
	case ULONG:
		return( SZLONG/SZINT );

	default:
		return(1);

		}
	}
/* ============================================== */

shltype( o, p ) NODE *p; {
	if( o == NAME|| o==REG || o == ICON || o == OREG ) return(1);
	return( o==UNARY MUL && shumul(p->left) );
	}
/* ---------------------------------------------- */

shumul( p ) register NODE *p; {
	register o;

	o = p->op;
	if( o == NAME || o == OREG || o == ICON ) return( STARNM );

	if( ( o == INCR || o == ASG MINUS ) &&
	    ( p->left->op == REG && p->right->op == ICON ) &&
	    p->right->name[0] == '\0' &&
	    spsz( p->left->type, p->right->lval ) )
		return( STARREG );

	return( 0 );
	}
/* ---------------------------------------------- */

special( p, shape ) register NODE *p; {
	/* special shape matching routine */

	switch( shape ) {

	case SCCON:
		if( p->op == ICON && p->name[0]=='\0' && p->lval>= -128
			&& p->lval <=127 ) return( 1 );
		break;

	case SICON:
		if( p->op == ICON && p->name[0]=='\0' && p->lval>= 0
			&& p->lval <=32767 ) return( 1 );
		break;

	default:
		cerror( "bad special shape" );

		}

	return( 0 );
	}
/* ---------------------------------------------- */

shtemp( p ) register NODE *p; {
	if( p->op == UNARY MUL ) p = p->left;
	if( p->op == REG || p->op == OREG )
		return( !istreg( p->rval ) );
	return( p->op == NAME || p->op == ICON );
	}
/* ---------------------------------------------- */

```

14.2 `lastchance` (4085)

This is called by `order` at line 1796, for no apparently good reason. However, there are apparently some situations (not for the PDP11) where one more try may be worthwhile.

14.3 `rewfld` (4091)

If there is any special hardware that can do part of the job of extracting bit fields from words, then this is the place to show it. Of course if hardware exists to handle the general case, `ffld` (1928) will never be invoked in the first place to call `rewfld`.

14.4 `spsz` (4096)

`spsz` is called by `deltest` (2947) and `shumul` (4147). Its function is to determine whether normal hardware autoincrement or autodecrement addressing, if used, will adjust a register pointer by the required amount.

14.5 `szty` (4126)

This procedure is called from several different procedures to determine the number of type A registers that will be needed to store a variable type. The answer is two for long integers, and one otherwise.

14.6 `shltype` (4141)

This procedure is called from several places (`gencall`, `match`, `setrew` and `stoasg`) to determine if the subtree has the shape of a "leaf" i.e. is directly addressable. This procedure could be virtually eliminated if addressing modes were handled more uniformly. Note also that, in the way the procedure is used, the first and second arguments are related, viz. `o == p->op`.

14.7 `shumul` (4147)

This procedure determines the shape (either `STARNM` or `STARREG`, or neither of these) of a tree that is known to be the subtree of a `UNARY MUL` operator. Trees of shape `STARNM` or `STARREG` correspond to standard PDP11 addressing modes, as discussed on pages 21 and 23. It is called by `tshape` at line 2317, and by several other procedures.

14.8 `special` (4163)

`special` is called by `tshape` at line 2262 to look for machine dependent shapes, particularly constants that may be capable of special treatment.

14.9 `shtemp` (4187)

`shtemp` is called by `tshape` at line 2268 to determine if the shape of the current subtree is consistent with a value in temporary (stack) storage. It doesn't actually have to be in the stack, provided it is not occupying, directly or indirectly, one of the temporary registers.

14.10 `flshape` (4195)

This procedure is called by `tshape` at line 2284 to determine whether the subtree of a `FLD` operator (which is `UTYPE`) is ready for generation of the field operation. (It may contain expressions that still need to be evaluated.)

14.11 `acon` (4202)

This procedure is called by `adrput`, `conput` and `upput` (all in this file), to insert the value of a constant into the assembler code stream. Since nodes of type `ICON` may represent address constants or arithmetic constants, these have to be distinguished.

4205: `CONFMT` is defined as `"L%d"` so that numeric values are generated in terms of their decimal equivalents. (Since the regular C compiler emits constants in octal, this is one point where the outputs of the two compilers are noticeably different. In fact, for some simple programs, this, and the numbering of labels, are almost the only differences.)

```
flshape( p ) register NODE *p; {
	register o = p->op;
	if( o==NAME || o==REG || o==ICON || o==OREG ) return( 1 );
	return( o==UNARY MUL && shumul(p->left)==STARNM );
	}
/* ================================================= */

acon( p ) register NODE *p; { /* print out a constant */

	if( p->name[0] == '\0' ){	/* constant only */
		printf( CONFMT, p->lval);
		printf( "." );
		}
	else if( p->lval == 0 ) {	/* name only */
		printf( "%.8s", p->name );
		}
	else {				/* name + offset */
		printf( "%.8s+", p->name );
		printf( CONFMT, p->lval );
		printf( "." );
		}
	}
/* ------------------------------------------------- */

adrcon( val ) CONSZ val; {
	printf( CONFMT, val );
	}
/* ------------------------------------------------- */

adrput( p ) register NODE *p; {
	/* output an address, with offsets, from p */

	if( p->op == FLD ){
		p = p->left;
		}
	switch( p->op ){

	case NAME:
		acon( p );
		return;

	case ICON:
		/* addressable value of the constant */
		if( szty( p->type ) == 2 ) {
			/* print the high order value */
			CONSZ save;
			save = p->lval;
			p->lval = ( p->lval >> SZINT ) & BITMASK(SZINT);
			printf( "$" );
			acon( p );
			p->lval = save;
			return;
			}
		printf( "$" );
		acon( p );
		return;

	case REG:
		printf( "%s", rnames[p->rval] );
		return;

	case OREG:
		if( p->rval == R5 ){  /* in the argument region */
			if( p->name[0] != '\0' ) werror( "bad arg temp" );
			printf( CONFMT, p->lval );
			printf( ".(r5)" );
			return;
			}
		if( p->lval != 0 || p->name[0] != '\0' ) acon( p );
		printf( "(%s)", rnames[p->rval] );
		return;

```

4208: Unmodified address constant.

4212: Observe the '+' sign.

14.12 `adrcon` (4219)

Not much to say here. Called only by `expand` at line 2410 to output a bitmask for use in a field operation.

14.13 `adrput` (4224)

`adrput` is called primarily by `expand` at line 2436 to expand the code character `A` It is also called by `eprint` at line 1154 and `zzzcode` at line 4603. The discussion on addressing modes in Chapter Two should be read in conjunction with this procedure.

4227: "De-reference" an initial `FLD` operator, if any.

4238: Insert a literal constant into the assembler code stream. If the constant type is `LONG` or `ULONG` (as reported by `szty` (4126)) emit the high order part of the constant only here. The other half will be handled appropriately by `upput` (in due course or has already been so handled). This involves reducing the value of `lval`, and subsequently restoring it.

4257: In the case of an `OREG`, if the associated register is `R5`, then we are dealing with a stack location, so there had better not be an associated name. Either the variable is an unnamed temporary variable, or it is an argument or named variable, whose name should have been suppressed in the first pass.

4263: Emit a constant and/or a symbolic name, if appropriate.

4264: Emit the reference to the register (as a pointer or an indexed pointer).

4269: Generate the indirection symbol at the beginning of the address.

4273: Finally, take care of autoincrementing and autodecrementing. Alter the tree to look like an `OREG` to fool `adrput` (4224), and also so that `reclaim` will find the result in a correctly addressable form.

14.14 `conput` (4309)

`conput` is called by `expand` at line 2438, and also by `zzzcode` at lines 4556 and 4563. From `expand`, it is used to expand the code character `C` occurring in a matched template. Such characters occur at lines 5448 and 5460 in connection with the initialization of data storage. At line 5468, there is a branch instruction, which should only be generated by Fortran programs.

The calls to `conput` from `zzzcode` originate in the templates for `bit`, `ash` and `ashc` instructions (see lines 5052, 5131 and 5262 respectively).

14.15 `insput` (4326)

This procedure is null for the PDP11 version. It has a use in the Honeywell version of the Portable C compiler for generating references to machine registers.

14.16 `upput` (4331)

This procedure, which is called by `expand` at line 2440 to expand the code character `U`, complements `adrput`, by handling "the other half" for long operands. It should be compared and contrasted with `adrput` (4224).

```
	case UNARY MUL:
		/* STARNM or STARREG found */
		if( tshape(p, STARNM) ) {
			printf( "*" );
			adrput( p->left);
			}
		else {	/* STARREG - really auto inc or dec */
			/* turn into OREG so replacement node will
			   reflect the value of the expression */
			register i;
			register NODE *q, *l;

			l = p->left;
			q = l->left;
			p->op = OREG;
			p->rall = q->rall;
			p->lval = q->lval;
			p->rval = q->rval;
			for( i=0; i<NCHNAM; i++ )
				p->name[i] = q->name[i];
			if( l->op == INCR ) {
				adrput( p );
				printf( "+" );
				p->lval -= l->right->lval;
				}
			else {	/* l->op == ASG MINUS */
				printf( "-" );
				adrput( p );
				}
			tfree( l );
		}
		return;

	default:
		cerror( "illegal address" );
		return;

		}

	}
/* ------------------------------------------------------ */

conput( p ) register NODE *p; {
	switch( p->op ){

	case ICON:
		acon( p );
		return;

	case REG:
		printf( "%s", rnames[p->rval] );
		return;

	default:
		cerror( "illegal conput" );
		}
	}
/* ------------------------------------------------------ */

insput( p ) NODE *p; {
	cerror( "insput" );
	}
/* ------------------------------------------------------ */
```

14.17 `rmove` (4378)

This procedure is called by `reclaim` at line 2786 to generate an explicit register-to-register move (integer or float). This is needed when the result calculated by an expression tree has been forced into the wrong register and it must be moved to the required register.

14.18 `hopcode` (4399)

`hopcode` is called by `expand` at line 2418 to output an operator name. See the comment at line 4400. Note that for floating point operations, the character `f` is appended to the operation name.

14.19 `zzzcode` (4415)

This procedure has been saved until last! It is highly specialized, and depends very much on the contents of `table`. It does all the dirty work (i.e. machine-dependent cases) for `expand`. The latter calls `zzzcode` when it has encountered a `'Z'` in the code string. The following character from the code string is passed as an argument to `zzzcode`.

The corresponding procedure for e.g. the VAX11/780 is vastly different from the code presented here. Since this procedure is likely to be rewritten on an ad hoc basis for any new version of the Portable C compiler, a detailed analysis is not appropriate

Some sampling may be in order however. It is suggested that the reader should at least look at the following:

4419: Generate byte versions of instructions when appropriate.

4434: Generate branch statements from within the templates.

4542: More references to `toff`.

4615: Structure assignment.

```

upput( p ) NODE *p; {
	/* output the address of the second word in the
	   pair pointed to by p (for LONGs)*/
	CONSZ save;

	if( p->op == FLD ){
		p = p->left;
		}

	save = p->lval;
	switch( p->op ){

	case NAME:
		p->lval += SZINT/SZCHAR;
		acon( p );
		break;

	case ICON:
		/* addressable value of the constant */
		p->lval &= BITMASK(SZINT);
		printf( "$" );
		acon( p );
		break;

	case REG:
		printf( "%s", rnames[p->rval+1] );
		break;

	case OREG:
		p->lval += SZINT/SZCHAR;
		if( p->rval == R5 ){  /* in the argument region */
			if( p->name[0] != '\0' ) werror( "bad arg temp" );
			}
		if( p->lval != 0 || p->name[0] != '\0' ) acon( p );
		printf( "(%s)", rnames[p->rval] );
		break;

	default:
		cerror( "illegal upper address" );
		break;

		}
	p->lval = save;

	}
/* ------------------------------------------------------ */

rmove( rt, rs, t ) TWORD t; {
	printf( "	%s	%s,%s\n", (t==FLOAT||t==DOUBLE)?
			"movf":"mov", rnames[rs], rnames[rt] );
	}
/* ------------------------------------------------------ */

struct hoptab { int opmask; char * opstring; } ioptab[]= {

	ASG PLUS,	"add",
	ASG MINUS,	"sub",
	ASG OR,		"bis",
	ASG AND,	"bic",
	ASG ER,		"xor",
	ASG MUL,	"mul",
	ASG DIV,	"div",
	ASG MOD,	"div",
	ASG LS,		"asl",
	ASG RS,		"asr",

	-1, ""	};

```

```
hopcode( f, o ){
	/* output the appropriate string from the above table */

	register struct hoptab *q;

	for( q = ioptab;  q->opmask>=0; ++q ){
		if( q->opmask == o ){
			printf( "%s", q->opstring );
			if( f == 'F' ) printf( "f" );
			return;
			}
		}
	cerror( "no hoptab for %s", opst[o] );
	}
/* ------------------------------------------------- */

zzzcode( p, c ) NODE *p; {
	register m;
	switch( c ){

	case 'B':	/* output b if type is byte */
		if( p->type == CHAR || p->type == UCHAR ) printf( "b" );
		return;

	case 'N':  /* logical ops, turned into 0-1 */
		/* use register given by register 1 */
		cbgen( 0, m=getlab(), 'I' );
		deflab( p->label );
		printf( "	clr	%s\n", rnames[getlr( p, '1' )->rval] );
		if( p->type == LONG || p->type == ULONG )
			printf( "	clr	%s\n",
				rnames[getlr( p, '1' )->rval + 1] );
		deflab( m );
		return;

	case 'I':
	case 'F':
		cbgen( p->op, p->label, c );
		return;

	case 'A':
	case 'C':
		/* logical operators for longs
		   defer comparisons until branch occurs */

		brnode = tcopy( p );
		brcase = c;
		return;

	case 'H':  /* fix up unsigned shifts */
		{	register NODE *q;
			register r, l;
			TWORD t;

			if( p->op == ASG LS ) return;
			if( p->op != ASG RS ) cerror( "ZH bad" );
			if( p->left->op != REG ) cerror( "SH left bad" );

			r = p->left->rval;
			t = p->left->type;
			l = (t==LONG || t == ULONG );

			if( t != UNSIGNED && t != UCHAR && t != ULONG )
				return;  /* signed is ok */

			/* there are three cases: right side is a constant,
				and has the shift value; right side is
				a temporary reg, and has the - shift value,
				and right side is something else: A1 has the
				- shift value then */

```

```
				/* in the case where the value is known (constant
				   rhs), the mask is just computed & put out... */

				if( p->right->op == ICON ){
					int s;
					s = p->right->lval;
					if( 1 ){
						if( s >= 16 ){
							printf( "	clr	r%d\n", r );
							s -= 16;
							++r;
							}
						}
					if( s >= 16 ) printf( "	clr	r%d\n", r );
					else {
						m = 0100000;
						m >>= s;  /* sign extends... */
						m <<= 1;
						printf( "	bic	$%o,r%d\n", m, r );
						}
					return;
					}

				/* general case */

				if( istnode( p->right ) ) q = p->right;
				else q = getlr( p, '1' );  /* where -shift
							   is stored */

				/* first, store the shifted value on the stack */
				printf( "	mov	r%d,-(sp)\n", r );
				if( 1 ) printf( "	mov	r%d,-(sp)\n", r+1 );

				/* now, make a mask */

				printf( "	mov	$100000,r%d\n", r );
				if( 1 ) printf( "	clr	r%d\n", r+1 );

				/* shift (arithmetically ) */
				if( 1 ) expand( q, RNOP, "	ashc	AR" );
				else expand( q, RNOP, "	ash	AR" );
				printf( ",r%d\n", r );

				if( 1 ) printf( "	ashc	$1,r%d\n", r );
				else printf( "	asl	r%d\n", r );

				/* now, we have a mask: use it to clear sp,
					and reload */
				if( 1 ){
					printf("\tbic\tr%d,(sp)\n\tmov\t(sp)+,r%d\n",
						r+1, r+1 );
					}
				printf("\tbic\tr%d,(sp)\n\tmov\t(sp)+,r%d\n",r,r);
				/* whew! */
				return;
				}

		case 'V':
			/* sign extend or not -- register is one less than the
			   left descendent */

			m = p->left->rval - 1;

			if( ISUNSIGNED(p->type) ){
				printf( "	clr	r%d\n", m );
				}
			else {
				printf( "	sxt	r%d\n", m );
				}
			return;
```

```
		/* stack management macros */
	case '-':
		if( toff ++ ) printf( "-" );
		printf( "(sp)" );
		return;

	case '4':
		if( toff == 0 ) ++toff; /* can't push doubles that way */
		printf( "-(sp)" );
		toff += 4;
		return;

	case '~':
		/* complemented CR */
		p->right->lval = ~p->right->lval;
		conput( getlr( p, 'R' ) );
		p->right->lval = ~p->right->lval;
		return;

	case 'M':
		/* negated CR */
		p->right->lval = -p->right->lval;
		conput( getlr( p, 'R' ) );
		p->right->lval = -p->right->lval;
		return;

	case 'L':  /* INIT for long constants */
		{
			unsigned hi, lo;
			lo = p->left->lval & BITMASK(SZINT);
			hi = ( p->left->lval >> SZINT ) & BITMASK(SZINT);
			printf( "	%o; %o\n", hi, lo );
			return;
		}

	case 'T':
		/* Truncate longs for type conversions:
		    LONG|ULONG -> CHAR|UCHAR|INT|UNSIGNED
		   increment offset to second word */

		m = p->type;
		p = p->left;
		switch( p->op ){
		case NAME:
		case OREG:
			p->lval += SZINT/SZCHAR;
			return;
		case REG:
			rfree( p->rval, p->type );
			p->rval += 1;
			p->type = m;
			rbusy( p->rval, p->type );
			return;
		default:
			cerror( "Illegal ZT type conversion" );
			return;

			}

	case 'U':
		/* same as AL for exp under U* */
		if( p->left->op == UNARY MUL ) {
			adrput( getlr( p->left, 'L' ) );
			return;
			}
		cerror( "Illegal ZU" );
		/* NO RETURN */
```

```
        case 'W':   /* structure size */
              if( p->op == STASG )
                    printf( "%d", p->stsize);
              else  cerror( "Not a structure" );
              return;

        case 'S':  /* structure assignment */
              {
                    register NODE *l, *r;
                    register size, count;

                    if( p->op == STASG ){
                          l = p->left;
                          r = p->right;
                          }
                    else if( p->op == STARG ){
                          /* store an arg onto the stack */
                          r = p->left;
                          }
                    else cerror( "STASG bad" );

                    if( r->op == ICON ) r->op = NAME;
                    else if( r->op == REG ) r->op = OREG;
                    else if( r->op != OREG ) cerror( "STASG-r" );

                    size = p->stsize;
                    count = size / 2;

                    r->lval += size;
                    if( p->op == STASG ) l->lval += size;

                    while( count-- ){ /* simple load/store loop */
                          r->lval -= 2;
                          expand( r, FOREFF, "    mov   AR," );
                          if( p->op == STASG ){
                                l->lval -= 2;
                                expand( l, FOREFF, "AR\n" );
                                }
                          else {
                                printf( "-(sp)\n" );
                                }

                          }

                    if( r->op == NAME ) r->op = ICON;
                    else if( r->op == OREG ) r->op = REG;

                    }
              break;

        default:
              cerror( "illegal zzzcode" );
              }
        }
/* ================================================== */
```

The last file, `table.c`, begins at line 4664 and consists merely of the initialization of the array `table`. This is an array of structures of type `optab` which was discussed earlier in Section 2.5. Each such structure defines a template that, when matched against a particular subtree, will result in the rewriting of the tree and the emission of zero, one, or more lines of assembly code.

To recapitulate briefly, each template specifies an operator, alternatives for the shape and type of each of the left and right subtrees and additional resources that may be needed during the code sequence. The last part of each template is a (pointer to a) character string, or *code string*, which, when expanded, becomes the string of instructions.

15.1 Macro Expansion

After a successful template match, the associated code string is expanded macro-fashion into assembler language statements, and the expression tree is rewritten to reflect the effects of the code which has been generated.

The style of macro expansion conducted by `expand` (2376) depends ultimately on the assembler for the target machine. As can be seen at line 4681 for example, upper case letters are used for macro names. Several of these are standardized and are recognized by `expand`. For example:

AL	address of the operand derived from the left subtree.
AR	address of the operand derived from the right subtree.
A1	address of the temporary operand (usually a register) assigned for the code sequence (may be the same as either AL or AR if either of the latter may be shared).
UR	address of the less significant word of a two-word operand derived from the left subtree.
U1	address of the less significant word of a two-word operand in a temporary location.
Z	first character of a machine-dependent macro. The character immediately following the Z is passed as an argument to `zzzcode`.

15.2 Table Searching

Searches of `table` are conducted in a linear fashion. Some of the overhead of conducting such searches has been removed by the obvious improvement of determining operator-dependent places from which to begin searching. This improvement, which takes advantage of the clustering of templates by operator type, is embodied in the procedure `setrew` (2112), and in corresponding changes to `match` (2159). This ensures that when a template match is made, the amount of searching involved is relatively limited. However the wasted effort can be considerable in the case where a match will not be made, since the search proceeds (fruitlessly) through the rest of the `table` until the appropriate one of the "catch-all" templates at the end (see lines 5482 to 5517) is encountered.

It will often be worthwhile to embark upon judicious reordering of the template groups so that those operator groups occur towards the end of `table` that result in relatively frequent unsuccessful matches*. Within each template group, the ones which lead to the most efficient object code should appear first. Another obvious improvement would be to move the more specific of the "catch-all" templates to earlier points in the table. For example, the template at line 5513

* According to Tom London, this has already been done for the VAX11/780 version of the compiler.

```
# include "mfile2"

# define AWD SNAME|SOREG|SCON|STARNM|STARREG|SAREG
# define LWD SNAME|SOREG|SCON|SAREG

struct optab  table[] = {

ASSIGN,       INAREG|FOREFF|FORCC,
       AWD,   TPOINT|TINT|TUNSIGNED|TCHAR|TUCHAR,
       SZERO,       TANY,
              0,    RLEFT|RRIGHT|RESCC,
              "  clrZB AL\n",

ASSIGN,       INAREG|FOREFF|FORCC,
       AWD,   TINT|TUNSIGNED,
       AWD,   TCHAR,
              NAREG|NASR, RLEFT|RESC1|RESCC,
              "  movb  AR,A1\n     mov   A1,AL\n",

ASSIGN,       INAREG|FOREFF|FORCC,
       AWD,   TINT|TUNSIGNED,
       AWD,   TUCHAR,
              0,    RLEFT|RESCC,
              "  movb  AR,AL\n     bic   $!377,AL\n",

ASSIGN,       INAREG|FOREFF|FORCC,
       AWD,   TPOINT|TINT|TUNSIGNED|TCHAR|TUCHAR,
       AWD,   TPOINT|TINT|TUNSIGNED|TCHAR|TUCHAR,
              0,    RLEFT|RRIGHT|RESCC,
              "  movZB AR,AL\n",

ASSIGN,       INAREG|FOREFF,
       LWD,   TLONG|TULONG,
       SZERO,       TANY,
              0,    RLEFT|RRIGHT,
              "  clr   AL\n  clr   UL\n",

ASSIGN,       INAREG|FOREFF,
       LWD,   TLONG|TULONG,
       LWD,   TLONG|TULONG,
              0,    RLEFT|RRIGHT,
              "  mov   AR,AL\n     mov   UR,UL\n",

ASSIGN,       FOREFF|INAREG,
       STARNM,          TLONG|TULONG,
       LWD,   TLONG|TULONG,
              NAREG|NASL, RRIGHT,
              "  mov   ZU,A1\n     mov   AR,(A1)+\n  mov   UR,(A1)\n",

ASSIGN,       FOREFF,
       STARNM,      TLONG|TULONG,
       AWD,   TUNSIGNED|TPOINT,
              NAREG|NASL, RRIGHT,
              "  mov   ZU,A1\n     clr   (A1)+\n     mov   AR,(A1)\n",

ASSIGN,       FOREFF,
       STARNM,      TLONG|TULONG,
       AWD,   TINT,
              NAREG|NASL, RRIGHT,
              "  mov   ZU,A1\n     mov   AR,2(A1)\n  sxt   (A1)\n",

       /* PANIC! */
ASSIGN,       FOREFF|INAREG,
       STARNM,      TLONG|TULONG,
       AWD,   TUNSIGNED|TPOINT,
              NAREG|NASL|NASR,  RESC1,
              "  mov  AR,-(sp)\n  mov  ZU,A1\n  clr  (A1)+\n\
                mov  (sp)+,(A1)\nF  mov  (A1),U1\nF  clr  A1\n",
```

for "ASG OPANY" could be moved to follow line 5305 after the ASG template.

15.3 Some Statistics

There are many ways to analyze the contents of `table`. The following summaries may be found of some assistance to the reader.

15.3.1 Template Operators. The accompanying table lists the operator for each template together with the line number at which it occurs. (The "catch-all" templates that begin at line 5484 are not included.)

```
4671 ASSIGN          4946 OPLOG            5212 ASG MINUS
4677 ASSIGN          4952 OPLOG            5218 ASG OR
4683 ASSIGN          4958 OPLOG            5225 ASG AND
4689 ASSIGN          4964 OPLOG            5231 ASG ER
4695 ASSIGN          4970 CCODES           5240 ASG ER
4701 ASSIGN          4976 CCODES           5246 ASG ER
4707 ASSIGN          4982 UNARY MINUS      5252 ASG LS
4713 ASSIGN          4988 UNARY MINUS      5258 ASG RS
4719 ASSIGN          4994 UNARY MINUS      5264 ASG RS
4726 ASSIGN          5000 COMPL            5270 ASG RS
4733 ASSIGN          5006 INCR             5276 ASG RS
4740 ASSIGN          5012 DECR             5282 ASG OPFLOAT
4746 ASSIGN          5018 INCR             5288 ASG OPFLOAT
4752 ASSIGN          5024 DECR             5294 ASG OPFLOAT
4758 ASSIGN          5030 INCR             5300 ASG OPFLOAT
4764 ASSIGN          5036 DECR             5306 UNARY CALL
4770 ASSIGN          5042 COMPL            5312 UNARY CALL
4776 ASSIGN          5048 AND              5318 SCONV
4782 ASSIGN          5054 ASG MUL          5324 SCONV
4788 ASSIGN          5060 ASG DIV          5330 SCONV
4794 ASSIGN          5066 ASG MOD          5336 SCONV
4800 ASSIGN          5072 ASG PLUS         5342 SCONV
4807 UNARY MUL       5078 ASG PLUS         5348 SCONV
4813 OPLTYPE         5084 ASG MINUS        5354 SCONV
4818 OPLTYPE         5090 ASG MINUS        5360 SCONV
4824 OPLTYPE         5096 ASG OR           5366 SCONV
4830 OPLTYPE         5103 ASG AND          5372 SCONV
4836 OPLTYPE         5109 ASG ER           5378 SCONV
4842 OPLTYPE         5115 ASG OPSHFT       5384 SCONV
4849 OPLTYPE         5121 ASG LS           5390 SCONV
4855 OPLTYPE         5127 ASG RS           5396 SCONV
4861 OPLTYPE         5133 ASG RS           5402 SCONV
4867 OPLTYPE         5139 ASG RS           5408 PCONV
4873 OPLTYPE         5145 ASG RS           5414 PCONV
4879 OPLTYPE         5151 ASG RS           5420 STARG
4885 OPLTYPE         5157 ASG OR           5426 STASG
4891 UNARY MUL       5164 ASG AND          5432 STASG
4897 OPLTYPE         5170 ASG PLUS         5438 STASG
4903 OPLTYPE         5176 ASG PLUS         5444 INIT
4909 OPLTYPE         5182 ASG PLUS         5450 INIT
4915 OPLTYPE         5188 ASG PLUS         5456 INIT
4921 OPLTYPE         5194 ASG MINUS        5464 GOTO
4927 OPLTYPE         5200 ASG MINUS        5470 GOTO
4934 OPLOG           5206 ASG MINUS        5476 GOTO
4940 OPLOG
```

```
ASSIGN,     FOREFF|INAREG,
        STARNM,     TLONG|TULONG,
        AWD,    TINT,
                NAREG|NASL|NASR,    RESC1,
                "   mov   AR,-(sp)\n   mov   ZU,A1\n   mov   (sp)+,2(A1)\n\
                F   mov   2(A1),U1\n   sxt   (A1)\nF   sxt   A1\n",

ASSIGN,     FOREFF|INAREG,
        STARNM,     TLONG|TULONG,
        SAREG,      TLONG|TULONG,
                0,      RRIGHT,
                "   mov   AR,AL\n   mov   ZU,AR\n   mov   UR,2(AR)\nF   mov   (AR),AR\n",

ASSIGN,     INAREG|FOREFF,
        LWD,    TLONG|TULONG,
        AWD,    TCHAR,
                NAREG,      RESC1,
                "   movb   AR,U1\n    mov   U1,UL\n   sxt   AL\nF sxt   A1\n",

ASSIGN,     INAREG|FOREFF,
        LWD,    TLONG|TULONG,
        AWD,    TUCHAR,
                0,      RLEFT,
                "   movb   AR,UL\n     bic   $!377,UL\n   clr   AL\n",

ASSIGN,     INAREG|FOREFF,
        LWD,    TLONG|TULONG,
        AWD,    TINT,
                0,      RLEFT,
                "   mov   AR,UL\n     sxt   AL\n",

ASSIGN,     INAREG|FOREFF,
        LWD,    TLONG|TULONG,
        AWD,    TUNSIGNED|TPOINT,
                0,      RLEFT,
                "   mov   AR,UL\n     clr   AL\n",

ASSIGN,     INBREG|INTBREG|FOREFF,
        AWD,    TDOUBLE,
        SBREG,      TDOUBLE,
                0,      RRIGHT,
                "   movf  AR,AL\n",

ASSIGN,     INBREG|INTBREG|FOREFF,
        AWD,    TFLOAT,
        SBREG,      TDOUBLE,
                0,      RRIGHT,
                "   movfo AR,AL\n",

ASSIGN, INAREG|FOREFF,
        SFLD, TANY,
        SZERO,      TANY,
                0,      RRIGHT,
                "   bic   $M.,AL\n",

ASSIGN, INTAREG|INAREG|FOREFF,
        SFLD, TANY,
        STAREG,     TANY,
                0,      RRIGHT,
                "F  mov   AR,-(sp)\n   ash   $H.,AR\n   bic   $!M.,AR\n\
                  bic   $M.,AL\n   bis   AR,AL\nF   mov   (sp)+,AR\n",
ASSIGN, INAREG|FOREFF,
        SFLD, TANY,
        AWD,  TANY,
                NAREG,      RRIGHT,
                "   mov   AR,A1\n   ash   $H.,A1\n   bic   $!M.,A1\n\
                  bic   $M.,AL\n   bis   A1,AL\n",
ASSIGN,     FOREFF,
        AWD,    TFLOAT,
        AWD,    TFLOAT,
                NBREG,      RESC1,
                "   movof AR,A1\n     movfo A1,AL\n",
```

15.3.2 Operator Summary. The following table gives the various operators that can be matched together with their frequencies of occurrence.

```
 1 AND
 3 ASG AND
 1 ASG DIV
 4 ASG ER
 2 ASG LS
 6 ASG MINUS
 1 ASG MOD
 1 ASG MUL
 4 ASG OPFLOAT
 1 ASG OPSHFT
 3 ASG OR
 6 ASG PLUS
 9 ASG RS
22 ASSIGN
 2 CCODES
 2 COMPL
 3 DECR
 3 GOTO
 3 INCR
 3 INIT
 6 OPLOG
19 OPLTYPE
 2 PCONV
15 SCONV
 1 STARG
 3 STASG
 2 UNARY CALL
 3 UNARY MINUS
 2 UNARY MUL
```

15.3.3 Visit Summary. The following table lists the various purposes (associated with the idea of "cookie") for which templates may be used and that occur in `table`. The numbers give the frequency of occurrence for each "visit".

```
 6 FORARG
13 FORCC
11 FOREFF
 4 FOREFF|INAREG
26 INAREG
 9 INAREG|FORCC
 8 INAREG|FOREFF
 4 INAREG|FOREFF|FORCC
 5 INAREG|INTAREG
 2 INBREG|INTBREG
 2 INBREG|INTBREG|FOREFF
 9 INTAREG
14 INTAREG|INAREG
 7 INTAREG|INAREG|FOREFF
 7 INTBREG
 3 INTBREG|INBREG
 3 INTEMP
```

15.3.4 Shape Summary. The following table lists the various tree shapes that occur in `table`. The numbers give the frequency of occurrence for each shape.

```
72 AWD
38 LWD
60 SANY
17 SAREG
 2 SAREG|SNAME|SOREG|SCON
 7 SBREG
 3 SBREG|AWD
 1 SCCON
11 SCON
 2 SCON|SAREG
 3 SFLD
 2 SICON
 2 SNAME
 4 SNAME|SOREG
 5 SONE
15 STAREG
10 STARNM
 7 STBREG
 5 SZERO
```

15.4 Some Comments

There is a great deal that can be said about the details of this file. As the reader will now be aware, the contents of this file have to be read closely in conjunction with the contents of the the two machine-dependent files `order.c` and `local2.c`. Also some of the code which constitutes the final program is emitted in the first pass of the compiler, and the reader must turn to the files `code.c` and `local.c`, which are not discussed in this document, for details about these.

4666: AWD represents a combination of shapes which together constitute the concept of an "addressable word" or addressable operand.

4667: LWD represents a restricted version of AWD for operands which may be addressed directly without the aid of a temporary register. This is an appropriate shape for long operands.

```

/* put this here so UNARY MUL nodes match OPLTYPE when appropriate */
UNARY MUL,  INTAREG|INAREG,
        SANY, TANY,
        STARNM,     TLONG|TULONG,
                NAREG|NASR, RESC1,
                "   mov   AL,U1\n     mov   (U1)+,A1\n  mov   (U1),U1\n",

OPLTYPE,    FOREFF,
        SANY, TANY,
        LWD,  TANY,
                0,    RRIGHT,
                "",   /* throw away computations which don't do anything */
OPLTYPE,    INTAREG|INAREG,
        SANY, TANY,
        SZERO,      TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
                NAREG|NASR, RESC1,
                "   clr   A1\n",

OPLTYPE,    INTAREG|INAREG,
        SANY, TANY,
        SZERO,      TLONG|TULONG,
                NAREG|NASR, RESC1,
                "   clr   A1\n  clr   U1\n",

OPLTYPE,    INTAREG|INAREG,
        SANY, TANY,
        SANY, TINT|TUNSIGNED|TPOINT|TCHAR,
                NAREG|NASR, RESC1,
                "   movZB AR,A1\n",

OPLTYPE,    INTEMP,
        SANY, TANY,
        SANY, TINT|TUNSIGNED|TPOINT,
                NTEMP,      RESC1,
                "   mov   AR,A1\n",

OPLTYPE,    FORCC,
        SANY, TANY,
        SANY, TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
                0,    RESCC,
                "   tstZB AR\n",


OPLTYPE,    FORARG,
        SANY, TANY,
        SANY, TINT|TUNSIGNED|TPOINT,
                0,    RNULL,
                "   mov   AR,Z-\n",

OPLTYPE,    INTAREG|INAREG,
        SANY, TANY,
        AWD,  TUCHAR,
                NAREG|NASR, RESC1,
                "   movb  AR,A1\n     bic   $!377,A1\n",

OPLTYPE,    INTAREG|INAREG,
        SANY, TANY,
        LWD,  TLONG|TULONG,
                NAREG,      RESC1,
                "   mov   UR,U1\n     mov   AR,A1\n",

OPLTYPE,    INTAREG|INAREG,  /* for use when there are no free regs */
        SANY, TANY,
        LWD,  TLONG|TULONG,
                NAREG|NASR, RESC1,
                "   mov   AR,-(sp)\n  mov   UR,U1\n     mov   (sp)+,A1\n",

OPLTYPE,    INTEMP,
        SANY, TANY,
        LWD,  TLONG|TULONG,
                2*NTEMP,    RESC1,
                "   mov   AR,A1\n     mov   UR,U1\n",
```

4671: The first template represents a simple instruction pair (`clr` and `clrb`) that can be used for a variety of purposes. It can be used to clear a register ("`INAREG`"), or to zero a word or byte in memory without leaving a result in a register ("`FOREFF`"), or to set the condition codes ("`FORCC`"). In the latter case, the result will be available in the condition codes ("`RESCC`"); otherwise it can be found at the address of either the right or left operand.

4675: The string `"ZB"` is reduced by `zzzcode` (see line 4419) to either the single character `"b"`, or to nothing, denoting a character or a word instruction respectively.

4677: This template can be used for the same general purposes as the previous template. It assigns a character to a word in two stages. A byte is moved into a temporary register, and then the content of this register is moved to the destination*. The intermediate register may be an unused temporary register ("`NAREG`"), or it may be the same as the register used to address the right operand ("`NASR`"), if the content of that register is not needed for another purpose. As before, the result of the operation may be found in the condition codes ("`RESCC`") if the "cookie" was `FORCC`. If the "cookie" was `FOREFF`, then there is of course no result to be found and saved.

4730: Notice the explicit use of the `sp` register at this point.

4737: In the interests of compactness, not readability, the tab characters in the code string have been removed.

4782: Three templates for field assignments begin here.

4813: `OPLTYPE` represents operations on leaves, mostly for movement from one location to another, but also for type conversion. The first template says that any such operation which is being performed `FOREFF` is always a null operation.

4849: This template moves a single word represented by a "leaf" node into the stack to satisfy the "cookie" `FORARG`.

4970: The code generated by this template is more complex than the code string suggests at first glance. Code to expand `ZN` found on line 4974 can be found beginning at line 4423.

5054: The group of `ASG` operators begins here. Note that because of the two address instructions of the PDP11, there are in fact no templates for unadorned binary operations alone. For example, there are a pair of templates for `ASG PLUS`, but none for `PLUS` alone†. If any attempt is made to match the operator `PLUS`, a match will be made by the template at line 5515. (Because of preparation performed by `setrew` (2112), this will be the first template examined, not the last!)

5102: See the discussion for `hardops` (3802) in Chapter Thirteen. The comment on this line is no longer quite accurate.

---

* If this second move is redundant as would occur if the destination were a register, then an efficient "optimizer" should be able to eliminate it. Actually, Lee Benoy suggests that this case will not occur because an earlier match of the template found at line 4830 would be made. This has not been verified.

† This need not be so for machines like the VAX11/780 that have three address instructions. However it is difficult to exploit the potential of such machines fully so long as the compiler does not construct and manipulate ternary trees as well as binary trees.

```
OPLTYPE,    FORCC,
        SANY,  TANY,
        LWD,   TLONG|TULONG,
               0,     RESCC,
               "ZA",

OPLTYPE, FORARG,
        SANY,  TANY,
        LWD,   TLONG|TULONG,
               0,     RNULL,
               "  mov   UR,Z-\n     mov   AR,Z-\n",

UNARY MUL,  FORARG,
        STARNM,      TANY,
        SANY,  TLONG|TULONG,
               NAREG|NASR, RNULL,
               "  mov   AL,A1\n     mov   2(A1),Z-\n  mov   (A1),Z-\n",

OPLTYPE, FORARG,
        SANY,  TANY,
        SBREG,       TDOUBLE,
               0,     RNULL,
               "  movf  AR,Z4\n",

OPLTYPE,    INTBREG|INBREG,
        SANY,  TANY,
        AWD,   TDOUBLE,
               NBREG,       RESC1,
               "  movf  AR,A1\n",

OPLTYPE,    INTEMP,
        SANY,  TANY,
        SBREG,       TDOUBLE,
               4*NTEMP,     RESC1,
               "  movf  AR,A1\n",

OPLTYPE,    FORCC,
        SANY,  TANY,
        AWD,   TDOUBLE,
               0,     RESCC,
               "  tstf  AR\n  cfcc\n",

OPLTYPE,    INTBREG|INBREG,
        SANY,  TANY,
        AWD,   TFLOAT,
               NBREG,       RESC1,
               "  movof AR,A1\n",

OPLTYPE,    FORCC,
        SANY,  TANY,
        AWD,   TFLOAT,
               NBREG,       RESCC,
               "  movof AR,A1\n     cfcc\n",


OPLOG,      FORCC,
        AWD,   TPOINT|TINT|TUNSIGNED,
        AWD,   TPOINT|TINT|TUNSIGNED,
               0,     RESCC,
               "  cmp   AL,AR\nZI",

OPLOG,      FORCC,
        AWD,   TCHAR|TUCHAR,
        AWD,   TCHAR|TUCHAR,
               0,     RESCC,
               "  cmpb  AL,AR\nZI",

OPLOG,      FORCC,
        AWD,   TCHAR|TUCHAR,
        SCCON,       TINT,   /* look for constants between -128 and 127 */
               0,     RESCC,
               "  cmpb  AL,AR\nZI",
```

```

OPLOG,      FORCC,
        LWD,    TLONG|TULONG,
        LWD,    TLONG|TULONG,
                0,      RESCC,
                "ZCZI",

OPLOG,      FORCC,
        SBREG,      TDOUBLE,
        AWD,    TFLOAT,
                NBREG,      RESCC,
                "   movof AR,A1\n    cmpf  A1,AL\n    cfcc\nZF",

OPLOG,      FORCC,
        SBREG,      TDOUBLE,
        SBREG|AWD,  TDOUBLE,
                0,      RESCC,
                "   cmpf  AR,AL\n    cfcc\nZF",

CCODES,     INTAREG|INAREG,
        SANY,   TANY,
        SANY,   TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
                NAREG,      RESC1,
                "   mov   $1,A1\nZN",

CCODES,     INTAREG|INAREG,
        SANY,   TANY,
        SANY,   TLONG|TULONG,
                NAREG,      RESC1,
                "   clr   A1\n  mov   $1,U1\nZN",

UNARY MINUS,        INTAREG|INAREG,
        STAREG,     TINT|TUNSIGNED,
        SANY,   TANY,
                0,      RLEFT,
                "   neg   AL\n",

UNARY MINUS,        INTAREG|INAREG,
        STAREG,     TLONG|TULONG,
        SANY,   TANY,
                0,      RLEFT,
                "   neg   AL\n  neg   UL\n  sbc   AL\n",

UNARY MINUS,        INTBREG|INBREG,
        STBREG,     TDOUBLE,
        SANY,   TANY,
                0,      RLEFT,
                "   negf  AL\n",

COMPL,      INTAREG|INAREG,
        STAREG,     TINT|TUNSIGNED,
        SANY,   TANY,
                0,      RLEFT,
                "   com   AL\n",

INCR,   INTAREG|INAREG|FOREFF,
        AWD,    TINT|TUNSIGNED|TPOINT,
        SONE,   TANY,
                NAREG,      RESC1,
                "F    mov   AL,A1\n    inc   AL\n",

DECR,   INTAREG|INAREG|FOREFF,
        AWD,    TINT|TUNSIGNED|TPOINT,
        SONE,   TANY,
                NAREG,      RESC1,
                "F    mov   AL,A1\n    dec   AL\n",

INCR,   INTAREG|INAREG|FOREFF,
        AWD,    TINT|TUNSIGNED|TPOINT,
        SCON,   TANY,
                NAREG,      RESC1,
                "F    mov   AL,A1\n    add   AR,AL\n",
```

```

DECR,	INTAREG|INAREG|FOREFF,
	AWD,	TINT|TUNSIGNED|TPOINT,
	SCON,	TANY,
		NAREG,		RESC1,
		"F	mov	AL,A1\n		sub	AR,AL\n",

INCR,	INTAREG|INAREG|FOREFF,
	LWD,	TLONG|TULONG,
	SCON,	TANY,
		NAREG,		RESC1,
		"F mov AL,A1\nF mov UL,U1\n add AR,AL\n add UR,UL\n adc AL\n",

DECR,	INTAREG|INAREG|FOREFF,
	LWD,	TLONG|TULONG,
	SCON,	TANY,
		NAREG,		RESC1,
		"F mov AL,A1\nF mov UL,U1\n sub AR,AL\n sub UR,UL\n sbc AL\n",

COMPL,		INTAREG|INAREG,
	STAREG,		TLONG|TULONG,
	SANY,	TANY,
		0,	RLEFT,
		"	com	AL\n	com	UL\n",

AND,	FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT,
	SCON,	TANY,
		0,	RESCC,
		"	bit	AL,$Z~\n",

ASG MUL,	INAREG,
	STAREG,		TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		NAREG,		RLEFT,
		"	mul	AR,AL\n",

ASG DIV,	INAREG,
	STAREG,		TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		NAREG,		RESC1,
		"ZV	div	AR,r0\n",	/* since lhs must be in r1 */

ASG MOD,	INAREG,
	STAREG,		TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		NAREG,		RLEFT,
		"ZV	div	AR,r0\n",	/* since lhs must be in r1 */

ASG PLUS,	INAREG|FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
	SONE,	TINT,
		0,	RLEFT|RESCC,
		"	incZB AL\n",

ASG PLUS,	INAREG|FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		0,	RLEFT|RESCC,
		"	add	AR,AL\n",

ASG MINUS,	INAREG|FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
	SONE,	TINT,
		0,	RLEFT|RESCC,
		"	decZB AL\n",

ASG MINUS,	INAREG|FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		0,	RLEFT|RESCC,
		"	sub	AR,AL\n",
```

```

ASG OR,		INAREG|FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		0,	RLEFT|RESCC,
		"	bis	AR,AL\n",

/* AND transformed to "pdp11 bic"  in first pass. */
ASG AND,	INAREG|FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		0,	RLEFT|RESCC,
		"	bic	AR,AL\n",

ASG ER,		INAREG|FORCC,
	AWD,	TINT|TUNSIGNED|TPOINT,
	SAREG,		TINT|TUNSIGNED|TPOINT,
		0,	RLEFT|RESCC,
		"	xor	AR,AL\n",

ASG OPSHFT,		INAREG,
	SAREG,		TINT|TUNSIGNED|TPOINT,
	SONE,	TINT,
		0,	RLEFT,
		"	OI	AL\nZH",

ASG LS,		INAREG,
	SAREG,		TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		0,	RLEFT,
		"	ash	AR,AL\n",

ASG RS,		INAREG,
	SAREG,		TINT|TUNSIGNED|TPOINT,
	SCON,	TANY,
		0,	RLEFT,
		"	ash	$ZM,AL\nZH",

ASG RS,		INAREG,
	SAREG,		TINT|TUNSIGNED|TPOINT,
	STAREG,		TINT|TUNSIGNED|TPOINT,
		0,	RLEFT,
		"	neg	AR\n	ash	AR,AL\nZH",

ASG RS,		INAREG,
	SAREG,		TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		NAREG|NASR,	RLEFT,
		"	mov	AR,A1\n	neg	A1\n	ash	A1,AL\nZH",

ASG RS,		INAREG,
	SAREG,		TINT,
	AWD,	TINT,
		0,	RLEFT,
		"	mov	AR,-(sp)\n	neg	(sp)\n	ash	(sp)+,AL\nZH",

ASG RS,		INAREG,
	SAREG,		TINT|TUNSIGNED|TPOINT,
	AWD,	TINT|TUNSIGNED|TPOINT,
		NTEMP,		RLEFT,
		"	mov	AR,A1\n	neg	A1\n	ash	A1,AL\nZH",

ASG OR,		INAREG|FORCC,
	AWD,	TCHAR|TUCHAR,
	AWD,	TCHAR|TUCHAR,
		0,	RLEFT|RESCC,
		"	bisb	AR,AL\n",

```

```
/* AND transformed to "pdp11 bic"  in first pass. */
ASG AND,     INAREG|FORCC,
        AWD,  TCHAR|TUCHAR,
        AWD,  TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
              0,     RLEFT|RESCC,
              "  bicb  AR,AL\n",

ASG PLUS,    INAREG,
        LWD,  TLONG|TULONG,
        SICON,       TINT|TLONG|TULONG,
              0,     RLEFT,
              "  add   UR,UL\n     adc   AL\n",

ASG PLUS,    INAREG,
        STARNM,      TLONG|TULONG,
        LWD,  TLONG|TULONG,
              NAREG,       RLEFT,
              "  mov ZU,A1\n  add AR,(A1)+\n add UR,(A1)\n  adc -(A1)\n",

ASG PLUS,    INAREG,
        LWD,  TLONG|TULONG,
        LWD,  TLONG|TULONG,
              0,     RLEFT,
              "  add   AR,AL\n     add   UR,UL\n     adc   AL\n",

ASG PLUS,    INAREG,
        AWD,  TPOINT,
        LWD,  TLONG|TULONG,
              0,     RLEFT,
              "  add   UR,AL\n",

ASG MINUS,   INAREG,
        LWD,  TLONG|TULONG,
        SICON,       TINT|TLONG|TULONG,
              0,     RLEFT,
              "  sub   UR,UL\n     sbc   AL\n",

ASG MINUS,   INAREG,
        STARNM,      TLONG|TULONG,
        LWD,  TLONG|TULONG,
              NAREG,       RLEFT,
              "  mov  ZU,A1\n   sub  AR,(A1)+\n sub  UR,(A1)\n  sbc  -(A1)\n",

ASG MINUS,   INAREG,
        LWD,  TLONG|TULONG,
        LWD,  TLONG|TULONG,
              0,     RLEFT,
              "  sub   AR,AL\n     sub   UR,UL\n     sbc   AL\n",

ASG MINUS,   INAREG,
        AWD,  TPOINT,
        LWD,  TLONG|TULONG,
              0,     RLEFT,
              "  sub   UR,AL\n",

ASG OR,      INAREG,
        LWD,  TLONG|TULONG,
        LWD,  TLONG|TULONG,
              0,     RLEFT,
              "  bis   AR,AL\n     bis   UR,UL\n",

/* AND transformed to "pdp11 bic"  in first pass. */
ASG AND,     INAREG,
        LWD,  TLONG|TULONG,
        LWD,  TLONG|TULONG,
              0,     RLEFT,
              "  bic   AR,AL\n     bic   UR,UL\n",
```

```
ASG ER,         INAREG,
        LWD,    TLONG|TULONG,
        SAREG,          TLONG|TULONG,
                0,      RLEFT,
                "  xor   AR,AL\n      xor   UR,UL\n",

        /* table entries for ^ which correspond to the usual way of doing
                business  (rhs in a temp register) */

ASG ER,         INAREG|INTAREG,
        STAREG,         TLONG|TULONG,
        LWD,    TLONG|TULONG,
                0,      RLEFT,
                "  mov   AL,-(sp)\n  mov  UR,AL\n\
                   xor  AL,UL\n  mov  AR,AL\n  xor  AL,(sp)\n  mov  (sp)+,AL\n",
ASG ER,         INAREG|INTAREG,
        STAREG,         TINT|TUNSIGNED|TPOINT,
        AWD,    TINT|TUNSIGNED|TPOINT,
                0,      RLEFT,
                "  mov AL,-(sp)\n  mov AR,AL\n  xor AL,(sp)\n  mov (sp)+,AL\n",

ASG LS,         INAREG,
        SAREG,          TLONG|TULONG,
        AWD,    TINT|TUNSIGNED|TPOINT,
                0,      RLEFT,
                "  ashc  AR,AL\n",

ASG RS,         INAREG,
        SAREG,          TLONG|TULONG,
        SCON,   TANY,
                0,      RLEFT,
                "  ashc  $ZM,AL\nZH",

ASG RS,         INAREG,
        SAREG,          TLONG|TULONG,
        STAREG,         TINT|TUNSIGNED|TPOINT,
                0,      RLEFT,
                "  neg   AR\n  ashc  AR,AL\nZH",

ASG RS,         INAREG,
        SAREG,          TLONG|TULONG,
        AWD,    TINT|TUNSIGNED|TPOINT,
                NAREG|NASR,  RLEFT,
                "  mov   AR,A1\n     neg   A1\n  ashc  A1,AL\nZH",

ASG RS,         INAREG,
        SAREG,          TLONG|TULONG,
        AWD,    TINT|TUNSIGNED|TPOINT,
                NTEMP,          RLEFT,
                "  mov   AR,A1\n     neg   A1\n  ashc  A1,AL\nZH",

ASG OPFLOAT,            INBREG|INTBREG,
        STBREG,         TDOUBLE,
        SBREG|AWD,      TDOUBLE,
                0,      RLEFT|RESCC,
                "  OF    AR,AL\n",

ASG OPFLOAT,            INBREG|INTBREG,
        STBREG,         TDOUBLE,
        AWD,    TFLOAT,
                NBREG|NBSR,  RLEFT|RESCC,
                "  movof AR,A1\n     OF    A1,AL\n",

ASG OPFLOAT,            FORCC,
        STBREG,         TDOUBLE,
        SBREG|AWD,      TDOUBLE,
                0,      RESCC,
                "  OF    AR,AL\n     cfcc\n",

ASG OPFLOAT,            FORCC,
        STBREG,         TDOUBLE,
        AWD,    TFLOAT,
                NBREG|NBSR,  RESCC,
                "  movof AR,A1\n     OF    A1,AL\n     cfcc\n",
```

```

UNARY CALL, INTAREG,
        SAREG|SNAME|SOREG|SCON, TANY,
        SANY, TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR|TLONG|TULONG,
                NAREG|NASL, RESC1, /* should be register 0 */
                "   jsr    pc,*AL\n",

UNARY CALL, INTBREG,
        SAREG|SNAME|SOREG|SCON, TANY,
        SANY, TDOUBLE|TFLOAT,
                NBREG,      RESC1, /* should be register FR0 */
                "   jsr    pc,*AL\n",

SCONV,          INTAREG,
        STAREG,         TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
        SANY, TUCHAR,
                0,      RLEFT,
                "   bic    $!377,AL\n",

SCONV,          INTAREG,
        AWD,    TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
        SANY, TCHAR|TINT,
                NAREG|NASL, RESC1,
                "   movZB AL,A1\n",

SCONV,          INAREG|INTAREG,
        LWD,    TLONG|TULONG,
        SANY, TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
                0,      RLEFT,
                "ZT",

SCONV,          INTAREG,
        AWD,    TUCHAR,
        SANY, TLONG|TULONG,
                NAREG|NASL, RESC1,
                "   movb  AL,U1\n     bic    $!377,U1\n  clr    A1\n",

SCONV,          INTAREG,
        AWD,    TINT,
        SANY, TLONG|TULONG,
                NAREG|NASL, RESC1,
                "   mov    AL,U1\n     sxt    A1\n",

SCONV,          INTAREG,
        AWD,    TUNSIGNED|TPOINT,
        SANY, TLONG|TULONG,
                NAREG|NASL, RESC1,
                "   mov    AL,U1\n     clr    A1\n",

SCONV,          INTAREG,
        SBREG,          TDOUBLE,
        SANY, TINT|TUNSIGNED|TPOINT|TCHAR|TUCHAR,
                NAREG,      RESC1,
                "   movfi AL,A1\n",

SCONV,          INTAREG,
        STBREG,         TDOUBLE,
        SANY, TLONG|TULONG,
                NAREG,      RESC1,
                " setl\n movfi AL,-(sp)\n seti\n mov (sp)+,A1\n mov (sp)+,U1\n",

SCONV,          FORARG,
        STBREG,         TDOUBLE,
        SANY, TLONG|TULONG,
                0,      RNULL,
                "   setl\n      movfi AL,Z4\n     seti\n",

SCONV,          INTBREG,
        SAREG,          TLONG,
        SANY, TANY,
                NBREG,      RESC1,
                "   mov   UL,-(sp)\n  mov   AL,-(sp)\n  setl\n\
```

```
                movif  (sp)+,A1\n  seti\n",
SCONV,          INTBREG,
        LWD,    TLONG,
        SANY,   TANY,
                NBREG,          RESC1,
                "  setl\n  movif  AL,A1\n  seti\n",

SCONV,          INTBREG,
        AWD,    TINT,
        SANY,   TANY,
                NBREG,          RESC1,
                "  movif AL,A1\n",

SCONV,          INTBREG,
        SAREG,          TULONG,
        SANY,   TANY,
                NBREG,          RESC1,
                "  mov  UL,-(sp)\n  mov  AL,-(sp)\n  setl\n  movif  (sp)+,A1\n\
                seti\n  cfcc\n  bpl  1f\n  addf  $050200,A1\n1:\n",
SCONV,          INTBREG,
        LWD,    TULONG,
        SANY,   TANY,
                NBREG,          RESC1,
                "  setl\n  movif  AL,A1\n  seti\n  cfcc\n  bpl  1f\n\
                addf  $050200,A1\n1:\n",
SCONV,          INTBREG,
        STAREG,         TUNSIGNED|TPOINT,
        SANY,   TANY,
                NBREG,          RESC1,
                "  movif  AL,A1\n  cfcc\n  bpl  1f\n  addf  $044200,A1\n1:\n",

PCONV,          INTAREG,
        AWD,    TCHAR|TUCHAR,
        SANY,   TPOINT,
                NAREG|NASL, RESC1,
                "  movb  AL,A1\n",

PCONV,          INAREG|INTAREG,
        LWD,    TLONG|TULONG,
        SANY,   TPOINT,
                0,      RLEFT,
                "ZT",

STARG,          FORARG,
        SNAME|SOREG,            TANY,
        SANY,   TANY,
                0,      RNULL,
                "ZS",

STASG,          FOREFF,
        SNAME|SOREG,            TANY,
        SCON|SAREG, TANY,
                0,      RNOP,
                "ZS",

STASG,          INTAREG|INAREG,
        SNAME|SOREG,            TANY,
        STAREG,         TANY,
                0,      RRIGHT,
                "ZS",

STASG, INAREG|INTAREG,
        SNAME|SOREG,            TANY,
        SCON|SAREG, TANY,
                NAREG,          RESC1,
                "ZS   mov   AR,A1\n",

INIT, FOREFF,
        SCON, TANY,
        SANY, TINT|TUNSIGNED|TPOINT,
                0,      RNOP,
                "  CL\n",
```

```

INIT, FOREFF,
	SCON, TANY,
	SANY, TLONG|TULONG,
		0,	RNOP,
		"ZL",

INIT, FOREFF,
	SCON, TANY,
	SANY, TCHAR|TUCHAR,
		0,	RNOP,
		"	.byte CL\n",

	/* for the use of fortran only */

GOTO, FOREFF,
	SCON, TANY,
	SANY, TANY,
		0,	RNOP,
		"	jbr	CL\n",

GOTO, FOREFF,
	SNAME,		TLONG|TULONG,
	SANY, TANY,
		0,	RNOP,
		"	jmp	*UL\n",

GOTO, FOREFF,
	SNAME,		TINT|TUNSIGNED|TCHAR|TUCHAR|TPOINT,
	SANY, TANY,
		0,	RNOP,
		"	jmp	*AL\n",

	/* Default actions for hard trees ... */

# define DF(x) FORREW,SANY,TANY,SANY,TANY,REWRITE,x,""

UNARY MUL, DF( UNARY MUL ),

INCR, DF(INCR),

DECR, DF(INCR),

ASSIGN, DF(ASSIGN),

STASG, DF(STASG),

OPLEAF, DF(NAME),

OPLOG,		FORCC,
	SANY, TANY,
	SANY, TANY,
		REWRITE,	BITYPE,
		"",

OPLOG,		DF(NOT),

COMOP, DF(COMOP),

INIT, DF(INIT),

OPUNARY, DF(UNARY MINUS),


ASG OPANY, DF(ASG PLUS),

OPANY, DF(BITYPE),

FREE, FREE, FREE, FREE, FREE, FREE, FREE, FREE, "help: I'm in trouble\n"  };
```

Chapter 16: Conclusion

It will be impossible for the reader to have reached this point without having formulated, as the writer has done, some definite opinions about the state of the second pass of the Portable C compiler in the PDP11 version.

First, it must be agreed that the Portable C compiler is a significant achievement: it does exist, it does work, it has been ported to several diverse computer species, and the effort to do so is bounded. For the PDP11, the code generated does not suffer unduly in comparison with that of the highly tuned production compiler. The source code which is examined in this document is neither excessively long nor excessively opaque.

While the Portable C compiler is a significant milestone along the road to portable code generation, it is not the end of the road. Nor have its authors made such claims. It is a springboard from which next major leap forward can be made.

Even though grand strategy for code generation used by the Portable C compiler is clear enough, the tactics in particular situations are often convoluted and unobvious. The grand design has become overburdened with special cases. A person who would implement a new version of the compiler has a task which is not straightforward or even easily specified. There are really very few tables which can be initialized in mechanical fashion to specify the characteristics of the target machine. Even `table`, the array of templates, is a hand-crafted expansion of the information contained in the processor handbook.

At many points in the code, the author's intent is difficult to fathom. At many points, the reader must ask himself: is this something that had to done? or could only be done this way? or did it seem like a good idea at the time? or is it an important, heuristically determined optimization? In too many cases, the answer is not clear, and the reader is left wondering in many situations why certain cases are accorded special attention, when apparently equally undeserving cases seem to be ignored entirely. Can it be shown that the latter cases will never happen? If so, the evidence is often very deeply buried.

This is a criticism which hardly confined to the Portable C compiler, but which can be leveled at perhaps the majority of programs, which have undergone extensive refinement and development since their original conception.

In these pages are a number of suggestions for detailed improvements to the present program. However the real gains will come from a major reexamination of the problems with code generation in the light of the experience already gained with the Portable C compiler. The next generation program should maintain much more information in tabular form; should provide for the mechanical generation of templates from much more condensed manually provided information; and should provide easier-to-use mechanisms for recognizing and handling subtree species.

In the opinion of the present writer, the continued development of the Portable C compiler is essential, even long after the present document will have become obsolete. The several versions of the Portable C compiler which already exist are generating centrifugal forces, which, if not restrained by the centralizing forces of a strong, continuing development of the compiler, will destroy one of its principal achievements, namely a family of *consistent* compilers for a single language.

Appendix A. Cross-reference

Variables that are only referenced once, and many variables whose name is a single letter such as p have been omitted from this listing.

```
A           2435 3992 4439
ALDOUBLE    0261 2656 2668
ALSTACK     0266 1011
AND         0012 0750 0761 0762 1768 1954 3218 3895 3902 4389 5048 5103 5164 5225
ANDAND      0017 0765 1213 1245 1362 1459 1575 1647 1872 3076 3247
ARS         0105 0797
ARY         0183 0203
ASG         0128 0756 0758 0760 0762 0771 0773 0775 0777 0779 0781 1720 1727 1735 1773 3030 3031 3032 3209 3210 3211 3216 3217 3218 3219 3408 3409 3410 3419 3464 3790 3791 3792 3796 3797 3798 3902 4153 4386 4387 4388 4389 4390 4391 4392 4393 4394 4395 4453 4454 5054 5060 5066 5072 5078 5084 5090 5096 5103 5109 5115 5121 5127 5133 5139 5145 5151 5157 5164 5170 5176 5182 5188 5194 5200 5206 5212 5218 5225 5231 5240 5246 5252 5258 5264 5270 5276 5282 5288 5294 5300 5513
ASGFLG      0142 0157 0756 0758 0760 0762 0769 0771 0773 0775 0777 0779 0781 0782 0783 0802 0803 0804 2084 2086 2088 2090 2096 2100 2102 2149 2186
ASGOPFLG    0152 0756 0758 0760 0762 0771 0773 0775 0777 0779 0781 0802 0803 2096
ASSIGN      0074 0769 1741 1753 3025 3158 3453 4671 4677 4683 4689 4695 4701 4707 4713 4719 4726 4733 4740 4746 4752 4758 4764 4770 4776 4782 4788 4794 4800 5492
AUTOINIT    0269 3759
AWD         4666 4672 4678 4679 4684 4685 4690 4691 4715 4721 4728 4735 4748 4754 4760 4766 4771 4777 4796 4801 4802 4857 4905 4917 4923 4929 4935 4936 4941 4942 4947 4960 4966 5007 5013 5019 5025 5049 5056 5062 5068 5073 5079 5080 5085 5091 5092 5097 5098 5104 5105 5110 5123 5141 5147 5153 5158 5159 5165 5166 5189 5213 5248 5254 5272 5278 5284 5290 5296 5302 5325 5337 5343 5349 5385 5409

B           2421 4419
BACKTEMP    0298 2652
BITMASK()   3697 4242 4350 4570 4571
BITOOR()    0331 2530
BITYPE      0138 0694 0709 0755 0756 0757 0758 0759 0760 0761 0762 0763 0764 0765 0766 0767 0768 0769 0770 0771 0772 0773 0774 0775 0776 0777 0778 0779 0780 0781 0782 0783 0784 0785 0786 0787 0788 0789 0790 0791 0792 0793 0794 0795 0796 0797 0798 0799 0800 0801 0802 0803 0804 0805 1104 1128 1275 1439 1591 1758 2230 2932 3088 5501 5515
BTMASK      0189 0195 0196 0204 0205
BTYPE()     0196 2342
BYTEOFF()   0329 0330 2271 2423

C           2427 3993 3996 4002 4440
CALL        0086 0751 0785 1247 1248 1338 1348 1399 1426 1429 1578 1579 1684 1686 1687 3067 3140 3178 3872 4048 5306 5312
CALLFLG     0149 0159 0751 0752 0753 0785 0786 0805
CAST        0125 0209 0803
CBRANCH     0123 0799 1254 1367 1573 1620
CCODES      0108 0735 1652 2297 2715 4970 4976
CHAR        0165 0198 2347 2772 3273 3329 3413 3421 3607 4108 4420
CM          0072 0767 1395 3628 3671 3867
COLON       0016 0764 3250
COMMFLG     0145 0755 0756 0761 0762 0778 0779 0780 0781 2085 2086
COMOP       0079 0768 1218 1253 1356 1571 1606 1893 3251 5506
COMPL       0089 0739 3897 3908 3914 5000 5042
CONFMT      0274 2424 4205 4213 4220 4259
CONSZ       0273 0483 0565 1054 1057 1997 2379 3613 4096 4219 4240 4334

DECR        0091 0783 1260 3205 5012 5024 5036 5490
DECREF()    0205 0844 2335 4101
DELAYS      0505 0508 1180 1262
DF()        5484 5486 5488 5490 5492 5494 5496 5504 5506 5508 5510 5513 5515
DIV         0080 0770 0771 1766 3031 3054 3209 3256 3788 3791 3794 3797 4392 5060
DIVFLG      0146 0770 0771 0772 0773 2089 2090
DOUBLE      0170 2368 2775 3020 3212 3257 3271 3503 3675 4022 4118 4379
DSIZE       0209 0724 0725 2110 2127

EQ          0092 0787 1804 1820 1827 1838 1839 1916 3069 3965 3979 3994 3999 4001 4004 4010 4011
ER          0014 0780 0781 1770 3258 3419 4390 5109 5231 5240 5246
EXIT        0577 0578

F           2392 4010 4407 4435
FCON        0008 0734
FIELDOPS    1311 1925
FLD         0115 0744 1657 1937 2282 2985 3097 3441 3447 3460 3512 4227 4336
FLOAT       0169 2366 2775 3020 3135 3212 3257 3271 3341 3503 3533 3608 3675 4022 4115 4379
FLOFLG      0147 0755 0756 0757 0758 0759 0760 0770 0771 2099 2100
FORARG      0388 3664 3734 4849 4885 4891 4897 5366 5420
FORCC       0386 1851 1866 1914 2712 3996 4002 4671 4677 4683 4689 4842 4879
```

```
          4915 4927 4934 4940 4946 4952 4958
          4964 5048 5072 5078 5084 5090 5096
          5103 5109 5157 5164 5294 5300 5498
FORCE          0122 0740 1572 1614 3021 3081
FOREFF         0381 1196 1198 1607 1703 1705
          1717 1784 1894 2393 2704 4642 4645
          4671 4677 4683 4689 4695 4701 4707
          4713 4719 4726 4733 4740 4746 4752
          4758 4764 4770 4776 4782 4788 4794
          4800 4813 5006 5012 5018 5024 5030
          5036 5426 5444 5450 5456 5464 5470
          5476
FORREW         0389 2169 4077 4081 5484
FORTCALL       0087 0752 0786 1251 1252 1339
          1349 1428 1431 1582 1583 1677 1679
          1680 0320 3021 4022
FREE           0109 0645 0658 0670 0678 0684
          1224 1266 1268 1611 1623 1624 1644
          1706 1789 1877 1885 1890 1895 1908
          1909 2117 2129 2172 2694 2765 3467
          3468 3630 3638 3826 3909 5517 5517
FTN            0182 0202 3879

GE             0096 0791 1804 1824 1843 3072
          3979
GOTO           0045 0745 5464 5470 5476
GT             0097 0792 1804 1825 1844 3071
          3967 3968 3969 3970 3979

H              2400 4448

I              1635 1852 1857 1868 1915 1916
          2431 4425 4434
ICON           0007 0733 1150 1832 1956 1974
          2032 2053 2146 2183 2255 2278 3107
          3369 3636 3831 3846 3877 3897 3905
          4142 4151 4154 4169 4174 4191 4197
          4236 4312 4348 4473 4630 4653
INAREG         0382 1664 1849 3370 3374 3414
          3424 3487 3499 3518 3536 3558 3571
          3575 3590 3731 3732 4045 4080 4671
          4677 4683 4689 4695 4701 4707 4726
          4733 4740 4746 4752 4758 4764 4782
          4788 4794 4807 4818 4824 4830 4855
          4861 4867 4970 4976 4982 4988 5000
          5006 5012 5018 5024 5030 5036 5042
          5054 5060 5066 5072 5078 5084 5090
          5096 5103 5109 5115 5121 5127 5133
          5139 5145 5151 5157 5164 5170 5176
          5182 5188 5194 5200 5206 5212 5218
          5225 5231 5240 5246 5252 5258 5264
          5270 5276 5330 5414 5432 5438
INBREG         0384 1664 1849 3424 3487 3499
          3504 3518 3536 3558 3571 3575 3590
          3731 3732 4080 4770 4776 4903 4921
          4994 5282 5288
INCR           0090 0782 1259 1712 1720 1727
          1728 2949 3204 3458 3461 4153 4287
          5006 5018 5030 5488 5490
INCREF()       0204 3832 3837 3879
INIT           0124 0741 1673 5444 5450 5456
          5508
INT            0167 0199 0200 0300 1941 2354
          2772 2844 2845 2848 2922 2923 2926
          3220 3274 3411 3848 3869 4111
INTAREG        0383 1617 1630 1637 1654 1670
          1749 1750 3370 3374 3385 3391 3474
          3475 3536 3540 3558 3571 3575 3590
          3595 3730 3735 4049 4078 4078 4080
          4788 4807 4818 4824 4830 4855 4861
          4867 4970 4976 4982 4988 5000 5006
          5012 5018 5024 5030 5036 5042 5240
          5246 5306 5318 5324 5330 5336 5342
          5348 5354 5360 5408 5414 5432 5438
INTBREG        0385 1617 1630 1637 1670 1749
          1750 3474 3475 3536 3540 3558 3571
          3575 3590 3595 3730 4049 4078 4080
          4770 4776 4903 4921 4994 5282 5288
          5312 5372 5378 5384 5390 5396 5402
INTEMP         0387 1358 1366 1413 2268 2977
          2980 2987 2990 2993 2997 3000 3536
          3558 3571 3575 3590 3733 4079 4080
          4836 4873 4909
ISARY()        0203 0848 2336
ISFTN()        0202 0847
ISPTR()        0201 0846 2333 3221 3274 3412
          4100 4103
ISUNSIGNED() 0197 4533

L              2191 2226 2413 2623 3697 4567
          4603
LB             0070 0798
LE             0094 0789 1804 1822 1841 3074
          3979
LOGFLG         0143 0158 0738 0765 0766 0787
          0788 0789 0790 0791 0792 0793 0794
          0795 0796 2097
LONG           0168 0198 1941 2364 3276 3277
          3335 3679 3787 3788 3789 3790 3791
          3792 3812 4130 4428 4459
LS             0082 0774 0775 1771 4394 4453
          5121 5252
LT             0095 0790 1804 1823 1842 3073
          3967 3968 3969 3970 3979
LTYPE          0136 0693 0729 0730 0731 0732
          0733 0734 0735 0736 1103 1127 1207
          1276 1334 1445 2093 2227 3087 3134
LWD            4667 4696 4702 4703 4709 4747
          4753 4759 4765 4815 4863 4869 4875
          4881 4887 4953 4954 5031 5037 5171
          5178 5183 5184 5190 5195 5202 5207
          5208 5214 5219 5220 5226 5227 5232
          5242 5331 5379 5397 5415

M              2404 2410 4560
MAXRVAR        0288 3750
MDONE          0376 1560 1793 2207 4051
MINRVAR        0289 3743 3750
MINUS          0010 0748 0757 0758 1663 1720
          1727 1728 1764 2008 2032 2037 2053
          2056 2060 3217 3368 3410 3464 4153
          4387 4982 4988 4994 5084 5090 5194
          5200 5206 5212 5510
MNOPE          0375 1562 2209
MOD            0081 0772 0773 1767 3032 3055
          3210 3256 3789 3792 3795 3798 4393
          5066
MUL            0011 0749 0759 0760 1344 1345
          1658 1702 1765 1999 2315 2633 2955
          2975 2986 3030 3053 3101 3103 3145
          3211 3256 3258 3279 3281 3282 3285
          3338 3366 3390 3437 3441 3448 3480
          3496 3507 3512 3533 3607 3644 3787
          3790 3793 3796 3825 4143 4188 4198
          4267 4391 4602 4807 4891 5054 5486
MULFLG         0150 0759 0760 0770 0771 2087
          2088
MUSTDO         0449 1171 1173 1633 2559 2776
          2778 3014 3021 3036 3042 3043 3048
          3049 3056 3063 3064 3082
MYREADER() 0348 1031

N              2405 4423
NACOUNT        0436 2500
```

```
NAMASK      0437 2502 2563 2590 2595
NAME        0005 0729 1151 1667 2146 2183
       2270 2276 2956 3389 3640 3830 4142
       4151 4191 4197 4232 4343 4584 4630
       4653 5496
NAREG       0435 2505 4680 4710 4716 4722
       4729 4736 4749 4797 4810 4821 4827
       4833 4858 4864 4870 4894 4973 4979
       5009 5015 5021 5027 5033 5039 5057
       5063 5069 5142 5179 5203 5273 5309
       5327 5339 5345 5351 5357 5363 5411
       5441
NASL        0438 2623 4710 4716 4722 4729
       4736 5309 5327 5339 5345 5351 5411
NASR        0439 2624 4680 4729 4736 4810
       4821 4827 4833 4858 4870 4894 5142
       5273
NBCOUNT     0441 2509
NBMASK      0442 2511 2570 2593
NBREG       0440 2514 4803 4906 4924 4930
       4961 5291 5303 5315 5375 5381 5387
       5393 5399 5405
NBSL        0443 2623
NBSR        0444 2624 5291 5303
NCHNAM      0237 0472 0482 1120 1122 2069
       2763 2904 4285
NE          0093 0788 1804 1821 1838 1840
       1915 3070 3965 3966 3979
NESTCALLS   0352 1410
NIL         0245 0638 1290 1296 1592 1678
       1685 1692 2147 2735 2741
NODE        0240 0245 0473 0474 0508 0509
       0510 0511 0513 0638 0640 0643 0652
       0654 0666 0675 0682 0688 0699 0884
       0966 1088 1093 1134 1180 1183 1202
       1221 1233 1263 1281 1307 1325 1392
       1420 1451 1524 1529 1806 1928 1931
       1988 1992 1996 2159 2166 2213 2214
       2238 2376 2452 2493 2546 2582 2620
       2629 2677 2678 2679 2681 2839 2891
       2909 2910 2913 2947 2960 2968 3006
       3093 3122 3295 3318 3363 3378 3383
       3398 3401 3492 3525 3526 3604 3623
       3668 3699 3802 3805 3888 3891 3926
       4021 4026 4032 4075 4085 4091 4141
       4147 4163 4187 4195 4202 4224 4277
       4309 4326 4331 4415 4449 4617
NOPREF      0450 1107 1169 1957 1970 1975
       2560 2779 3012 3026 3037 3045 3059
       3078 3841 3847 3868 3878 3915
NOT         0088 0738 1243 1369 1463 1577
       1649 1888 3075 5504
NTEMP       0445 2527 4839 4876 4912 5154
       5279
NTMASK      0446 2518 2527

OFFSZ       0278 0553 0554 0555 0556 0875
       0877 0879 0880 3756
OPANY       0366 5513 5515
OPFLOAT     0368 5282 5288 5294 5300
OPLEAF      0365 5496
OPLOG       0367 4934 4940 4946 4952 4958
       4964 5498 5504
OPLTYPE     0370 4813 4818 4824 4830 4836
       4842 4849 4855 4861 4867 4873 4879
       4885 4897 4903 4909 4915 4921 4927
OPSHFT      0369 5115
OPSIMP      0360 2140 2145 2177 2182
OPUNARY     0364 5510
OR          0013 0778 0779 1769 3219 3258
       3409 4388 5096 5157 5218
OREG        0107 0732 1152 2066 2146 2183
       2270 2312 2519 2634 2790 2842 2920
       2956 3389 3646 3649 3835 4142 4151
       4189 4197 4256 4281 4359 4585 4631
       4632 4654
OROR        0018 0766 1214 1246 1363 1460
       1576 1648 1880 3077 3248

PCONV       0117 0743 5408 5414
PLUS        0009 0755 0756 1720 1727 1728
       1735 1763 2008 2016 2032 2033 2053
       3107 3216 3258 3368 3408 3840 4386
       5072 5078 5170 5176 5182 5188 5513
PMCONV      0118 0800
PTR         0181 0201 0204 0236 2844 2848
       2922 2926
PUTCHAR()   0586 0984 2385
PVCONV      0119 0801

QUEST       0015 0763 1212 1244 1364 1461
       1574 1627 1899 3249

            0306 3082 4022 0307 3036 3042 3056
            0310 3043 3063 3064 0590 2027 2028
            2047 0593 2635 2791 2843 2921 0591
            2636 2793 2844 2922 0592 2636 2794
            2845 2923 0311 3048 3049 3063 0315
            0335 4257 4361
R           2194 2229 2624 4556 4563
REG         0106 0528 0731 1116 1146 1639
       1669 2001 2016 2033 2034 2054 2300
       2501 2510 2640 2771 2841 2919 2949
       2956 3384 3420 3486 3495 3502 3517
       3836 4142 4154 4189 4197 4252 4316
       4355 4455 4588 4631 4654
REGLOOP()   0530 2465 2484 2536
REGSZ       0333 0530 2453 0458 2727 4680
       4729 4736 4749 4803 4810 4821 4827
       4833 4839 4858 4864 4870 4876 4906
       4912 4924 4973 4979 5009 5015 5021
       5027 5033 5039 5063 5309 5315 5327
       5339 5345 5351 5357 5363 5375 5381
       5387 5393 5399 5405 5411 5441 0459
       2728 0460 2729
RESCC       0461 2712 4674 4680 4686 4692
       4845 4882 4918 4930 4937 4943 4949
       4955 4961 4967 5051 5075 5081 5087
       5093 5099 5106 5112 5160 5167 5285
       5291 5297 5303
RETURN      0044 0802
REWRITE     0447 2118 2200 5484 5501
RLEFT       0456 2725 4674 4680 4686 4692
       4698 4704 4755 4761 4767 4985 4991
       4997 5003 5045 5057 5069 5075 5081
       5087 5093 5099 5106 5112 5118 5124
       5130 5136 5142 5148 5154 5160 5167
       5173 5179 5185 5191 5197 5203 5209
       5215 5221 5228 5234 5243 5249 5255
       5261 5267 5273 5279 5285 5291 5321
       5333 5417
RNOP        0462 2694 2823 3657 4509 4510
       5429 5447 5453 5459 5467 5473 5479
RNULL       0455 1036 1634 1725 1742 1785
       1869 1917 2694 2704 2818 3454 3659
       4006 4852 4888 4894 4900 5369 5423
RRIGHT      0457 2726 4674 4692 4698 4704
       4710 4716 4722 4743 4773 4779 4785
       4791 4797 4816 5435
RS          0083 0776 0777 1772 1969 4395
       4454 5127 5133 5139 5145 5151 5258
       5264 5270 5276
RTOLBYTES   0299 1944 2288
```

S 2396 4615

SANY 0394 2266 4808 4814 4819 4825 4831 4832 4837 4838 4843 4844 4850 4851 4856 4862 4868 4874 4880 4886 4893 4898 4904 4910 4916 4922 4928 4971 4972 4977 4978 4984 4990 4996 5002 5044 5308 5314 5320 5326 5332 5338 5344 5350 5356 5362 5368 5374 5380 5386 5392 5398 5404 5410 5416 5422 5446 5452 5458 5466 5472 5478 5484 5499 5500

SAREG 0395 2307 2309 3718 3719 3720 3721 3751 3751 4666 4667 4742 5111 5116 5122 5128 5134 5140 5146 5152 5233 5253 5259 5265 5271 5277 5307 5313 5373 5391 5428 5440

SBREG 0397 0526 2307 3723 3724 4772 4778 4899 4911 4959 4965 4966 5284 5296 5355

SCC 0399 2298

SCCON 4168 4948

SCON 0401 2279 3499 3732 4666 4667 5020 5026 5032 5038 5050 5129 5260 5307 5313 5428 5440 5445 5451 5457 5465

SCONV 0116 0742 5318 5324 5330 5336 5342 5348 5354 5360 5366 5372 5378 5384 5390 5396 5402

SETOFF() 0225 1011 2656 2668 3676 3680 3684 3688 3761

SETSTO() 0532 1358 1366 1413 2977 2980 2987 2990 2993 2997 3000

SFLD 0402 2283 4783 4789 4795

SHFFLG 0151 0774 0775 0776 0777 2101 2102

SHORT 0166 2349 2772

SICON 4173 5172 5196

SIMPFLG 0144 0755 0756 0757 0758 0761 0762 0778 0779 0780 0781 2083 2084

SMONE 1501 2254 2258

SNAME 0400 2277 3499 3732 3735 4666 4667 5307 5313 5421 5427 5433 5439 5471 5477

SONE 1500 2253 2257 5008 5014 5074 5086 5117

SOREG 0403 2313 3499 3732 3735 4045 4666 4667 5307 5313 5421 5427 5433 5439

SPECIAL 0344 0346 0407 0408 0409 0410 1498 1502 2248

SPFLG 0153 2103 2145 2182

STAREG 0396 0527 2309 2467 2564 3718 3719 3720 3751 4790 4983 4989 5001 5043 5055 5061 5067 5135 5241 5247 5266 5319 5403 5434

STARG 0111 0746 1109 1159 2521 2752 3633 3683 4624 5420

STARNM 0404 3508 3732 4151 4198 4269 4666 4708 4714 4720 4727 4734 4741 4809 4892 5177 5201

STARREG 0405 3448 3508 3534 3609 3732 4157 4666

STASG 0110 0804 1109 1160 1731 2522 3068 3183 4610 4620 4638 4643 5426 5432 5438 5494

STBREG 0398 0527 2309 2471 2571 3723 4995 5283 5289 5295 5301 5361 5367

STCALL 0112 0753 0805 1110 1157 1158 1249 1250 1340 1350 1402 1427 1430 1580 1581 1691 1693 1694 2521 2522 3141 3178

STOARG() 0337 1400

STOFARG() 0339 1406

STOSTARG() 0340 1403

STREF 0085 0784

STRING 0006 0730

STRTY 0171 2351

SWADD 0406 2270

SZCHAR 0250 1111 1112 2523 3760 4344 4360 4586

SZERO 0408 1499 2252 2256 4673 4697 4784 4820 4826

SZINT 0251 1947 2291 2524 2660 2666 4132 4242 4344 4350 4360 4570 4571 4586

SZLONG 0254 4132

TANY 0429 2328 4673 4697 4783 4784 4789 4790 4795 4796 4808 4814 4815 4819 4825 4831 4837 4843 4850 4856 4862 4868 4874 4880 4886 4892 4898 4904 4910 4916 4922 4928 4971 4977 4984 4990 4996 5002 5008 5014 5020 5026 5032 5038 5044 5050 5129 5260 5307 5313 5374 5380 5386 5392 5398 5404 5421 5422 5427 5428 5433 5434 5439 5440 5445 5451 5457 5465 5466 5472 5478 5484 5484 5499 5500

TBUSY 2451 2537 2603 2604 2610

TCHAR 0417 2348 4672 4679 4690 4691 4748 4820 4832 4844 4941 4942 4947 4972 5073 5085 5158 5159 5165 5166 5308 5319 5325 5326 5332 5356 5409 5458 5477

TDOUBLE 0422 2369 4771 4772 4778 4899 4905 4911 4917 4959 4965 4966 4995 5283 5284 5289 5295 5296 5301 5314 5355 5361 5367

TFLOAT 0421 2367 4777 4801 4802 4923 4929 4960 5290 5302 5314

TINT 0419 2355 4672 4678 4684 4690 4691 4721 4735 4760 4820 4832 4838 4844 4851 4935 4936 4948 4972 4983 5001 5007 5013 5019 5025 5049 5055 5056 5061 5062 5067 5068 5073 5074 5079 5080 5085 5086 5091 5092 5097 5098 5104 5105 5110 5111 5116 5117 5122 5123 5128 5134 5135 5140 5141 5146 5147 5152 5153 5166 5172 5196 5247 5248 5254 5266 5272 5278 5308 5319 5325 5326 5332 5343 5356 5385 5446 5477

TLONG 0420 2365 4696 4702 4703 4708 4709 4714 4720 4727 4734 4741 4742 4747 4753 4759 4765 4809 4826 4863 4869 4875 4881 4887 4893 4953 4954 4978 4989 5031 5037 5043 5171 5172 5177 5178 5183 5184 5190 5195 5196 5201 5202 5207 5208 5214 5219 5220 5226 5227 5232 5233 5241 5242 5253 5259 5265 5271 5277 5308 5331 5338 5344 5350 5362 5368 5373 5379 5415 5452 5471

TMASK 0186 0201 0202 0203

TMPREG 0335 2520

TNEXT() 0650 0657

TPOINT 0423 2342 4672 4690 4691 4715 4728 4766 4820 4832 4838 4844 4851 4935 4936 4972 5007 5013 5019 5025 5049 5055 5056 5061 5062 5067 5068 5073 5079 5080 5085 5091 5092 5097 5098 5104 5105 5110 5111 5116 5122 5123 5128 5134 5135 5140 5141 5152 5153 5166 5189 5213 5247 5248 5254

5266 5272 5278 5308 5319 5325 5332 5349 5356 5403 5410 5416 5446 5477
TPTRTO 0428 2333 2339 2343
TREESZ 0218 0220 0510 0563 0645 0650 0669
TSHIFT 0191 0204 0205
TSHORT 0418 2350
TSTRUCT 0430 2353
TUCHAR 0424 2361 4672 4685 4690 4691 4754 4820 4844 4857 4941 4942 4947 4972 5073 5085 5158 5159 5165 5166 5308 5319 5320 5325 5332 5337 5356 5409 5458 5477
TULONG 0427 2363 4696 4702 4703 4708 4709 4714 4720 4727 4734 4741 4742 4747 4753 4759 4765 4809 4826 4863 4869 4875 4881 4887 4893 4953 4954 4978 4989 5031 5037 5043 5171 5172 5177 5178 5183 5184 5190 5195 5196 5201 5202 5207 5208 5214 5219 5220 5226 5227 5232 5233 5241 5242 5253 5259 5265 5271 5277 5308 5331 5338 5344 5350 5362 5368 5391 5397 5415 5452 5471
TUNSIGNED 0426 2357 4672 4678 4684 4690 4691 4715 4728 4766 4820 4832 4838 4844 4851 4935 4936 4972 4983 5001 5007 5013 5019 5025 5049 5055 5056 5061 5062 5067 5068 5073 5079 5080 5085 5091 5092 5097 5098 5104 5105 5110 5111 5116 5122 5123 5128 5134 5135 5140 5141 5152 5153 5166 5247 5248 5254 5266 5272 5278 5308 5319 5325 5332 5349 5356 5403 5446 5477
TUSHORT 0425 2359
TWORD 0241 0470 0480 0489 0496 0821 2325 2854 2874 3605 3613 3783 3808 4096 4126 4378 4451
TYFLG 0141 0156 2092 2094 2098 2104
TYPE 0041 0736

U 2439 4600
UCHAR 0175 0197 2360 2773 3273 3330 3413 3422 3608 4107 4420 4461
UGE 0100 0794 1804 1818 1855 3969 3973 3979
UGT 0101 0793 1804 1819 1836 1838 3970 3971 3972 3973 3974 3979 3987
ULE 0098 0796 1804 1816 1837 3967 3971 3979
ULONG 0178 0197 2362 3276 3277 3336 3679 3793 3794 3795 3796 3797 3798 3812 4131 4428 4459 4461
ULT 0099 0795 1804 1817 1858 3968 3971 3972 3973 3974 3979
UNARY 0129 0748 0749 0750 0751 0752 0753 1110 1158 1248 1250 1252 1338 1339 1340 1344 1345 1426 1427 1428 1578 1580 1582 1658 1663 1677 1680 1684 1687 1691 1694 1702 1999 2315 2522 2633 2955 2975 2986 3101 3103 3140 3141 3145 3279 3281 3282 3285 3338 3366 3390 3437 3441 3448 3480 3496 3507 3512 3533 3607 3644 3825 4048 4143 4188 4198 4267 4602 4807 4891 4982 4988 4994 5306 5312 5486 5486 5510
UNIONTY 0172 2352
UNSIGNED 0177 0199 0200 2356 2774 3220 3274 3411 4112 4461
UPKFOFF() 0232 1945 1947 2289 2291
UPKFSZ() 0231 1943 2287
USHORT 0176 2358 2773
UTYPE 0137 0715 0738 0739 0740 0741 0742 0743 0744 0745 0746 0748 0749 0750 0751 0752 0753 1208 1375 1441 2091 2934 3138

ZCHAR 3118 3195 3196 3225 3305 3331
ZFLOAT 3120 3195 3196 3225 3305 3342
ZLONG 3119 3195 3196 3225 3305 3337

acon() 4202 4233 4244 4249 4263 4313 4345 4352 4364
adrcon() 2410 4219
adrput() 1154 2436 4224 4271 4288 4294 4603
again 1423 1444 1534 1618 1660 1665 1671 1710 1713 1721 1729 1732 1736 1751 1754 1759 1774 1796
allchk() 0582 1038 2479
allo() 2202 2493 0897 2458
argc 0890 0900 0961 0968 0974 0977
argsize() 3668 3672 4037
argv 0890 0901 0961 0968 0974 0978
asgop() 0157 1345 1380 1934 2973 3200 3822 4079
asop 3159 3221

base 1055 1070
baseoff 0555 0879 0992 0998 1003 1025 2662 2671
brcase 3700 3990 3996 4002 4015 4445
brnode 3699 3996 4002 4006 4016 4444
busy 0522 2308 2453 2466 2485 2537 2588 2599 2600 2601 2602 2603 2604 2609 2610 2781 2795 2796 2800 2863 2867 2882 2884

callchk() 0582 2701
callflag 0535 0886 1341 1353 1408 1411 1432
callop() 0159 2554 2699
callreg() 2555 4021
calltype 1392 1396 1399 1402
canon() 1289 1307 1538 1744 3472 3645 3648 3928
cbgen() 1635 1852 1857 1868 1915 1916 3981 4425 4436
cbranch() 1622 1628 1651 1806 1874 1875 1882 1883 1889 1896 1902 1903 1905
ccbranches 3935 3999 4001 4004 4011
cerror() 0607 0621 0660 0670 0683 0949 0995 1014 1043 1064 1077 1082 1113 1518 1604 2123 2233 2486 2586 2732 2748 2782 2797 2801 2863 2866 2868 2886 2981 3390 3462 3465 3578 3649 3859 3987 4179 4301 4321 4327 4369 4411 4454 4455 4595 4606 4612 4628 4632 4660
cform 0524 2739 2740
cleanup 1561 1612 1645 1655 1682 1689 1696 1780
cnames 1473 1506 1510
codgen() 1196 1198 1281 1607 1609 1630 1637 1849 1851 1866 1894 1914
conput() 2428 4309 4556 4563
constore() 1370 1451 1464
convert 3815 3821
cook 1524 1536 1596 1617 1703 1717 1784 1791 1793 1796 4085
cookie 1281 1302 1492 1498 1499 1500 1501 1502 1507 1528 1536 1543 1560 1563 1563 1598 1609 1681 1688 1695

2159 2169 2190 2205 2206 2376 2393 2677 2690 2704 2712 2740 4026 4028 4032 4075 4077 4078 4079
count 4618 4635 4640
cp 0894 0901 0902 0903 0906 0907 0949 0965 1020 1021 1995 2004 2019 2020 2024 2036 2038 2041 2045 2059 2060 2062 2065 2070 2376 2376 2381 2381 2382 2385 2389 2393 2410 2418 2422 2428 2432 2436 2440 3613
crslab 3351 3354
cstring 0548 2205

deflab() 1636 1638 1876 1884 1904 1906 1907 3358 4005 4426 4431
delay() 1035 1183 1219 1191 1202 1208 1216 1229 1195 1233 1275 1276
deli 0506 1181 1194 1198 1262 1264
deltest() 1261 2947
deltrees 0508 1180 1198 1264
dope 0156 0157 0158 0159 0246 0724 0815 2128 2149 2186
dopeop 0727 0814 0815 0816
dopest 0727 0812
dopeval 0727 0815 0700 0703 0705 0710 0717 1928 1934 3008 3014 3021 3026 3036 3078 3082 3087 0700 0703 0705 0712 1928 1935 3008 3012 3027 3037 3088
down 0699 0705 0712 0717 1134 1136 1137 1139 1141 1928 1937 3006 3010 3013 3014 3027

edebug 0574 0866 0915 1028 1291 1012 3755
eprint() 0567 1134
eprint 1028 1293 1545 1747
eread() 0515 1026 1089 1127 1128
expand() 2205 2376 3657 3996 4002 4509 4510 4642 4645

false 1806 1828 1829 1868 1873 1875 1876 1883 1889 1896 1900 1900 1905 1907 1916
ffld() 1312 1928
filename 0572 0860 1020 1022
files 0895 0898 0953 0957 0962 0968 0973 0974 0975 0977 0978 1048
flab 1810 1900 1903 1907
flag 1496 1505 1508 1509 2807 2828 2831 2832
fldshf 0573 2079 2289 2291 2401 2409
fldsz 0573 2079 2287 2291 2397 2407
flshape() 2284 4195
fltused 0349 2943 3022 3763 3764
fn 3770 3772
fop 3782 3814 3815
fprintf() 0605 0606 0614 0615 0616 0625 0629 0630 0631
freereg() 2502 2511 2546
freetemp() 2523 2527 2647
fregs 0537 0883 1358 1366 1382 3034 3045 3058 3063 3142 3180 3185 3213 3257 3743 3746 3749 3751
ftnno 0559 0861 0997 0999 3762
ftype 3783 3815
func 3784 3880
functbl 3781 3806
fwalk() 0699 0710 1028 1293 1313 1545 1747

genargs() 3623 3629 4041
gencall() 0341 1688 4028 4032
gencase 3214 3223
genfcall() 0341 1681
genscall() 1695 4026
getchar() 0980 0983 0994 1013 1020 1062 1113 1119
getlab() 1628 1635 1651 1873 1881 1900 1901 1902 3353 3995 4425
getlr() 0517 2191 2194 2214 2422 2428 2432 2436 2440 2632 4427 4430 4496 4556 4563 4603
gotit 2742 2750

hardops() 3802 3927
hi 4569 4571 4572
hopcode() 2418 4399
hoptab 4384 4402

indope 0727 0814
insput() 2432 4326
int 0688 0699
ioptab 4384 4404
isbreg() 0526 2307 2589
istnode() 0528 3265 3539 3557 3570 3574 3582 3594 4495
istreg() 0527 0528 2308 2485 2586 2598 2795 2796 2800 2862 2865 2882 2884 2885 2885 4190

3984 3995 4001 4005
lab 1810 1812 1873 1874 1876 1881 1882 1884 1902 1904 3981 3986 3999 4004 4011
label 0491 1653 1865 2414 4426 4436
lastchance() 1796 4085
lastfree 0640 0646 0656 0658
lbranches 3964 3994
leftadr 3427 3432 3486
lflag 0574 0864 0911 1022
lineid() 1022 3770
lineno 0571 0862 1019 1022
lo 4569 4570 4572
logop() 0158 1845 3188 3532 3549
lshape 0542 2192
lt 3262 3268 3271 3274 3276
ltype 0543 2193
lval 0483 1103 1621 1640 1832 1959 1962 1963 1978 2002 2035 2055 2058 2068 2256 2257 2258 2271 2422 2503 2512 2523 2527 2530 2530 2716 2761 2902 2950 3653 3656 3838 3850 3853 3881 3906 4156 4169 4170 4174 4175 4205 4208 4213 4241 4242 4242 4245 4259 4263 4283 4290 4340 4344 4350 4360 4364 4373 4475 4555 4557 4562 4564 4564 4570 4571 4586 4637 4638 4641 4644

1527 1635 1638
mamask 2082 2145 2182
markcall() 1357 1365 1420 1440 1462
mask 2241 2279 2280 2307 2309 2310
match() 1560 1793 2159 4049
max() 3115 3149 3162 3169 3173 3185 3197 3230 3236 3239 3239 3252 3252 3308 3312 3313
maxa 2454 2462 2468 2469 2564
maxb 2454 2462 2472 2473 2571
maxoff 0554 0877 0998 1003 1011 2661 2670 3758
maxtemp 0556 0880 1000 2662 2671
maxtreg 0558 0882 0993 3743

mform 0524 2742
min() 3116 3197 3312
mina 2454 2463 2468 2564
minb 2454 2463 2472 2571
mkadrs() 1383 2968
mkdope() 0811 0955
mkrall() 3043 3049 3064 3093
mode 3981 4010
more 0702 0713 0718 2120 2126
myreader() 0348 3926

name 0472 0482 1119 1833 1961 1979
2004 2019 2021 2024 2036 2042 2045
2059 2060 2062 2070 2255 2504 2513
2529 2764 2904 3849 3880 4155 4169
4174 4204 4209 4212 4258 4263 4286
4362 4364
ncopy() 1223 1267 1610 2891 2916
ndu 0240 0465
needs 0546 2118 2200 2497
negrel 1804 1827
nerrors 0244 0596 0603 0607 0623 0668
0977 1049
nextcook() 1563 4075
niceuty() 3553 3562 3566 3584 3604
node 0510 0645 0646 0650 0669
nomat 1564 1603 1669 1681 1688 1695
1733 1755 1776 1798
noswap 3265 3280 3283 3286 3303
notoff() 2065 3613
nr 3128 3131 3149 3162 3169 3173
3185 3194 3197 3213 3228 3230 3236
3236 3239 3252 3257 3308 3312 3313
nrecur 0561 0863 1034 1517

odebug 0574 0867 0919 1541 1594 1747
offstar() 1659 1709 3363 3438 3442 3481
3497 3509 3513 3535 3554 3563 3567
3585 3647
opfunc 3785 3814
opmask 4384 4404 4405
opmtemp 2144 2145 2149 2150 2181 2182
2186
opptr 2110 2153 2170
opst 0247 0725 0727 0816 1143 1599
1604 3578 3988 4411
opstring 4384 4406
optab 0539 2108 2110 2114 2165 2493
4669 3888 3929
opty 0689 0691 0693 0694
optype() 0156 0691 0707 1102 1206 1239
1332 1437 1549 2227 2230 2930 3016
3130
order() 1300 1302 1524 1654 1664 1670
1705 1749 1750 3370 3374 3385 3391
3414 3424 3474 3475 3487 3499 3504
3518 3536 3540 3558 3571 3575 3590
3595 3664 4045 1309 1315 1988
ormake 2005 2029 2048 2064

1529 1590 1607 1622 1628 1658 1659
1664 1724 1726 1727 1529 1591 1592
1608 1609 1610 1611 1621 1623 1629
1630 1633 1633 1634 1637 1641 1642
1643 1644 1740 1743 1749 1750 3401
3452 3455 3459 3474 3475 0890 0968
pc 1095 1119 1120 1122
plb 3983 3994 3998 3999 4000 4001
4004
popargs() 4050 4055
prcook() 1492 1543 1596 1598 2690
printf() 0846 0847 0848 0850 1138 1141
1143 1147 1153 1161 1162 1166 1168
1169 1171 1172 1173 1175 1292 1499
1500 1501 1502 1508 1510 1542 1544
1595 1597 1599 2245 2331 2397 2401
2414 2424 2687 2689 2691 2819 2824
2831 2833 2859 2879 3010 3359 3762
3765 3772 3986 3999 4001 4004 4011
4064 4067 4070 4205 4206 4209 4212
4213 4214 4220 4243 4248 4253 4259
4260 4264 4270 4289 4293 4317 4351
4356 4365 4379 4406 4407 4420 4427
4429 4478 4483 4488 4500 4501 4505
4506 4511 4513 4514 4519 4522 4534
4537 4543 4544 4549 4572 4611 4648

ql 1996 2009 2016 2017 2019 2027
2028 2033 2034 2042 2045 2047 2047
2054 2057 2058 2060 2062
qq 2678 2723 2725 2726 2727 2728
2729 2731 2735 2741
qr 1996 2010 2016 2017 2021 2024
2026 2027 2028 2032 2035 2036 2053
2054 2055 2059

2792 2793 2795 2792 2794 2796
radebug 0575 0869 0927 3010
rall 0469 0479 0488 0495 1107 1169
1171 1173 1539 1608 1629 1633 1745
1957 1970 1975 2559 2776 2778 2900
3013 3042 3048 3063 3098 3098 3104
3108 3473 3841 3847 3868 3878 3915
4282 4282
rallo() 1539 1745 3006 3045 3056 3059
3087 3088 3111 3473
rbusy() 1118 2784 2874 2919 2922 2923
2926 4592
rcount() 1516 1537 2168
rdebug 0575 0868 0923 2686 2858 2878
rdin() 0565 0990 0992 0993 1019 1055
1098 1103 1104 1106 1111 1112 2697
2839
reclaim() 1036 1634 1725 1742 1785 1869
1917 2206 2677 3454 3659 4006
recres 2681 2723 2731 2741
reread 0971 1048
resc 0511 2224 2452 2501 2502 2503
2504 2510 2511 2512 2513 2519 2520
2523 2527 2529 2530 2540 2727 2728
2729
respref 0524 2682 2739 3728 3729
revrel 3979 4010
rew 3423 3434 3445
rewfld() 1939 4091
rewrite 0547 2200 2206
rfree() 2785 2841 2844 2845 2848 2854
4589
rmove() 2786 4378
rnames 0569 1147 1173 2859 2879 3706
4253 4264 4317 4356 4365 4380 4380
4427 4430
rs 4378 4380
rshape 0544 2195
rstatus 0521 0526 0527 2467 2471 2564
2571 3717 3751
rt 3262 3269 3271 3273 3277 4378
4380
rtype 0545 2196
rval 0484 0528 1104 1118 1147 1633
1641 1942 1960 1977 2003 2027 2027
2028 2028 2047 2057 2067 2286 2307
2308 2502 2511 2520 2540 2635 2636
2638 2641 2717 2762 2780 2785 2786

2787 2791 2793 2794 2800 2840 2903
2918 3851 3882 4190 4253 4257 4264
4284 4317 4356 4361 4365 4427 4430
4457 4531 4589 4590 4592
rw 2677 2688 2694 2704 2712 2725
2726 2727 2728 2729 2806 2818 2823
2830
rwnames 2808 2829 2833
rwprint() 2688 2806
rwtable 2108 2119 2169

save 4240 4241 4245 4334 4340 4373
sdebug 0575 0870 0935 2244
setasg() 1754 3492
setasop() 1736 3398
setbin() 1759 3525
setincr() 1713 3378
setregs() 1007 3739
setrew() 0956 2112
setstr() 1732 3383
shape 2238 2245 2248 2250 2256 2257
2258 2262 2266 2268 2270 2277 2280
2283 2298 2310 2313 2317 4163 4166
shareit() 2601 2602 2615 2620
shltype() 2147 2184 2964 4044 4141
shp 1931 1968 1969 1970 1971 1972
1973 1974 1975 1976 1977 1978 1979
1980
shtemp() 2268 4187
shumul() 2317 3146 3448 3534 3609 4143
4147 4198
sign 1056 1059 1065 1071
size 3625 3635 3653 3655 4055 4058
4059 4060 4070 4618 4634 4635 4637
4638
special() 2262 4163
spoff 3756 3758 3759 3760 3761 3762
spsz() 2950 4096 4156
ssu 3296 3299
stalign 0499 1112 1162 3684
stderr 0605 0606 0614 0615 0616 0625
0629 0630 0631
stoarg() 1352 1392 1396
stoasg() 1345 1380 2960
stocook 0532 0533 0885 1300
store() 1295 1325 1351 1359 1376 1387
1388 1409 1414 1468
stotree 0509 0532 0884 1290 1296 1300
stsize 0498 1111 1161 2523 3635 3685
4611 4634
su 0471 0481 0490 0497 1175 1358
1366 1382 2974 2976 2979 2996 3132
3135 3142 3149 3156 3157 3162 3169
3173 3180 3185 3197 3228 3230 3236
3239 3252 3308 3312 3323 3325 3332
3332 3332 3338 3343 3347 3403 3404
3495 3531 3552 3552 3582
sucomp() 1309 3122
sul 3127 3156 3160 3169 3173 3185
3195 3197 3228 3230 3234 3239 3252
3265 3279 3285 3293 3299 3308 3312
3313 3400 3403 3420 3434
sur 3127 3157 3162 3166 3173 3185
3196 3197 3225 3226 3236 3239 3252
3265 3293 3299 3305 3306 3312 3313
3400 3404 3430
swap 3271 3274 3282 3288 3298
szty() 1941 1947 2017 2026 2054 2595
2641 2781 2859 2864 2879 2883 3131
3194 4126 4238

table 0550 2117 2129 4669
talloc() 0514 0653 1099 1955 1968 1973
2916 3839 3844 3866 3876 3913
tcheck() 0665 1039
tcopy() 0516 1264 1724 1740 2758 2910
2933 2935 3452 4444
tdebug 0575 0871 0931 2330
temp 0963 0990 0997 0999 1997 2002
2018 2035 2039 2055 2056 2058 2065
2068 4034 4037 4038 4050
tfree() 0675 1643 2071 2706 2714 2759
4296 0676 0678 0682
tinit() 0642 0671 0969
tlab 1810 1901 1903 1906
tmpoff 0553 0875 0992 1025 2656 2659
2660 2661 2662 2666 2668 2670 2670
2671 2672
tnames 0824 0850
toff 3702 3930 4058 4059 4543 4548
4550
tprint() 0821 1167
true 1806 1826 1828 1852 1857 1865
1875 1881 1883 1884 1889 1896 1901
1901 1905 1906 1915 1916
tshape() 1791 2192 2195 2238 2742 3508
4269
ttype() 2193 2196 2325 2339
tword 2325 2328 2331 2333 2339 2342
2343 2348 2350 2353 2355 2357 2359
2361 2363 2365 2367 2369
ty 1203 1206 1207 1208 1237 1239
1275 1276 1329 1332 1334 1375 1526
1549 1591 1932 1941 1952 1958 1971
1976 3008 3016 3087 3088 3127 3130
3134 3138
type 0470 0480 0489 0496 1106 1118
1167 1642 1941 1947 1952 1958 1971
1976 2017 2017 2026 2054 2065 2193
2196 2595 2641 2755 2772 2772 2772
2773 2774 2775 2781 2784 2785 2786
2841 2901 2901 2919 2950 3008 3017
3020 3131 3135 3194 3212 3212 3220
3221 3257 3268 3269 3327 3411 3411
3412 3413 3421 3422 3503 3533 3607
3675 3679 3679 3811 3832 3837 3842
3848 3869 3879 3879 3916 3916 4022
4156 4238 4420 4428 4458 4533 4581
4589 4591 4592
typedef 0240 0241

u 0604 0939
udebug 0574 0872 0940
uerror() 0599 1674
upput() 2440 4331
usable() 2556 2560 2565 2572 2582
ushare() 2623 2624 2629

val 0302 1057 1060 1064 1070 1072
1074 1084 2379 2406 2407 2408 2409
2410 2422 2423 2424 4219 4220
visit 0541 2190

walkf() 0678 0688 0693 0694 1315 1319
2697 3927 3929
werror() 0612 4258 4362
where() 0604 0613 0622 3776

xdebug 0574 0873 0945 3744 3746

zap 3318 3331 3337 3342
zum() 3195 3196 3225 3305 3318
zzzcode() 2389 4415

Appendix B. Defined Symbols

Symbols that are defined for the Second Pass of the Portable C Compiler are given here. Those that are not in fact used are flagged with an asterisk.

*	0258	ALCHAR	8
	0261	ALDOUBLE	16
*	0260	ALFLOAT	16
*	0259	ALINT	16
*	0262	ALLONG	16
*	0264	ALPOINT	16
*	0263	ALSHORT	16
	0266	ALSTACK	16
*	0265	ALSTRUCT	16
	0012	AND	14
	0017	ANDAND	23
*	0268	ARGINIT	32
*	0285	ARGREG	5
	0105	ARS	93
	0183	ARY	060
	0128	ASG	1+
	0142	ASGFLG	01
	0152	ASGOPFLG	020000
*	0031	ASOP	25
	0074	ASSIGN	58
	0269	AUTOINIT	48
	4666	AWD	SNAME|SOREG|SCON|STARNM|STARREG|SAREG
*	0297	BACKAUTO	
	0298	BACKTEMP	
*	0210	BCSZ	100 /* size of table to save break
	3697	BITMASK(n)	((1L<<n)-1)
	0331	BITOOR(x)	((x)>>3) /* bit offset to oreg offset */
	0138	BITYPE	010
*	0049	BREAK	41
	0189	BTMASK	017
*	0190	BTSHIFT	4
	0196	BTYPE(x)	(x&BTMASK) /* basic type of x */
	0329	BYTEOFF(x)	((x)&01)
	0086	CALL	70
	0149	CALLFLG	02000
*	0055	CASE	47
	0125	CAST	111
	0123	CBRANCH	109
	0108	CCODES	96
*	0281	CCTRANS(x)	x
	0165	CHAR	2
*	0042	CLASS	34
	0072	CM	56
	0016	COLON	22
	0145	COMMFLG	0100
	0079	COMOP	59
	0089	COMPL	77
	0274	CONFMT	"%Ld"
	0273	CONSZ	long
*	0050	CONTINUE	42
	0091	DECR	79
	0205	DECREF(x)	(((x>>TSHIFT)&~BTMASK)|(x&BTMASK))
*	0054	DEFAULT	46
	0505	DELAYS	20
*	0200	DEUNSIGN(x)	((x)+(INT-UNSIGNED))
	5484	DF(x)	FORREW,SANY,TANY,SANY,TANY,REWRITE,x,""
*	0213	DIMTABSZ	750 /* size of the dimension/size table */
	0080	DIV	60
	0146	DIVFLG	0200
*	0034	DIVOP	28
*	0052	DO	44
*	0084	DOT	68
	0170	DOUBLE	7
	0209	DSIZE	CAST+1 /* size of the dope array */
*	0047	ELSE	39
*	0057	ENUM	49
*	0300	ENUMSIZE(high,low)	INT
*	0173	ENUMTY	10

```
* 0199  ENUNSIGN(x)     ((x)+(UNSIGNED-INT))
  0092  EQ              80
* 0033  EQUOP           27
  0014  ER              19
* 0004  ERROR           1
  0578  EXIT            exit
  0164  FARG            1
  0008  FCON            5
  0115  FLD             103
  0169  FLOAT           6
  0147  FLOFLG          0400
* 0053  FOR             45
  0388  FORARG          020000  /* compute for an argument of a function */
  0386  FORCC           040   /* compute for condition codes only */
  0122  FORCE           108
  0381  FOREFF          01    /* compute for effects only */
  0389  FORREW          040000  /* search the table for a rewrite rule */
* 0087  FORTCALL        73
  0320  FR0             8
* 0321  FR1             9
* 0322  FR2             10
* 0323  FR3             11
* 0324  FR4             12
* 0325  FR5             13
  0109  FREE            97
  0182  FTN             040
  0096  GE              84
  0045  GOTO            37
  0097  GT              85
  0007  ICON            4
* 0046  IF              38
  0382  INAREG          02    /* compute into a register */
  0384  INBREG          010   /* compute into a lvalue register */
* 0036  INCOP           30
  0090  INCR            78
  0204  INCREF(x)       (((x&-BTMASK)<<TSHIFT)|PTR|(x&BTMASK))
  0124  INIT            110
  0167  INT             4
  0383  INTAREG         04    /* compute into a scratch register */
  0385  INTBREG         020   /* compute into a scratch lvalue register */
  0387  INTEMP          010000  /* compute into a temporary location */
  0203  ISARY(x)        ((x&TMASK)==ARY) /* is x an array type */
  0202  ISFTN(x)        ((x&TMASK)==FTN) /* is x a function type */
  0201  ISPTR(x)        ((x&TMASK)==PTR)
  0197  ISUNSIGNED(x)   ((x)<=ULONG&&(x)>=UCHAR)
* 0293  LABFMT          "L%d"
  0070  LB              54
* 0068  LC              52
  0094  LE              82
  0143  LOGFLG          020
  0168  LONG            5
* 0066  LP              50
  0082  LS              64
  0095  LT              83
* 0148  LTYFLG          01000
  0136  LTYPE           02
  4667  LWD             SNAME|SOREG|SCON|SAREG
  0288  MAXRVAR         4
  0376  MDONE           010001
  0289  MINRVAR         2
  0010  MINUS           8
  0375  MNOPE           010000
  0081  MOD             62
* 0195  MODTYPE(x,y)    x = (x&(-BTMASK))|y
* 0174  MOETY           11
  0011  MUL             11
  0150  MULFLG          04000
  0449  MUSTDO          010000  /* force register requirements */
* 0348  MYREADER(p)     myreader(p)
  0436  NACOUNT         03
  0437  NAMASK          017
  0005  NAME            2
  0435  NAREG           01
  0438  NASL            04    /* share left register */
```

```
  0439  NASR            010  /* share right register */
  0441  NBCOUNT         060
  0442  NBMASK          0360
  0440  NBREG           020
  0443  NBSL            0100
  0444  NBSR            0200
  0237  NCHNAM          8  /* number of characters in a name */
  0093  NE              81
  0352  NESTCALLS
* 0131  NOASG           (-1)+
* 0227  NOFIT(x,y,z)    ( (x%z + y) > z )
* 0235  NOLAB           (-1)
* 0450  NOPREF          020000  /* no preference for register assignment */
  0088  NOT             76
* 0132  NOUNARY         (-2)+
  0563  NRECUR          (10*TREESZ)
  0445  NTEMP           0400
  0446  NTMASK          07400
  0278  OFFSZ           long
  0366  OPANY           010014  /* any op... */
* 0361  OPCOMM          010002  /* +, &, |, ^ */
* 0363  OPDIV           010006  /* /, % */
  0368  OPFLOAT         010020  /* +, -, *, or / (for floats) */
  0365  OPLEAF          010012  /* leaves */
  0367  OPLOG           010016  /* logical ops */
  0370  OPLTYPE         010024  /* leaf type nodes (e.g, NAME, ICON) */
* 0362  OPMUL           010004  /* *, / */
  0369  OPSHFT          010022  /* <<, >> */
  0360  OPSIMP          010000  /* +, -, &, |, ^ */
  0364  OPUNARY         010010  /* unary ops */
  0013  OR              17
  0107  OREG            95
  0018  OROR            24
* 0214  PARAMSZ         100 /* size of the parameter stack */
* 0317  PC              7     /* program counter */
  0117  PCONV           105
* 0230  PKFIELD(s,o)    ((o<<6)|s)
  0009  PLUS            6
  0118  PMCONV          106
  0181  PTR             020
* 0586  PUTCHAR(x)      putchar(x)
  0119  PVCONV          107
  0015  QUEST           21
  0306  R0              0
  0307  R1              1
  0310  R2              2
  0590  R2PACK(x,y)     (0200*((x)+1)+y)
* 0593  R2TEST(x)       ((x)>=0200)
  0591  R2UPK1(x)       (((x)>>7)-1)
  0592  R2UPK2(x)       ((x)&0177)
  0311  R3              3
* 0312  R4              4
  0315  R5              5     /* frame pointer */
* 0071  RB              55
* 0069  RC              53
  0106  REG             94
  0530  REGLOOP(i)      for(i=0;i<REGSZ;++i)
  0333  REGSZ           14
* 0032  RELOP           26
  0458  RESC1           04
  0459  RESC2           010
  0460  RESC3           020
  0461  RESCC           04000
* 0104  RESETBIT        92
  0044  RETURN          36
  0447  REWRITE         010000
  0456  RLEFT           01
  0462  RNOP            010000  /* DANGER: can cause loops.. */
  0455  RNULL           0       /* clobber result */
* 0067  RP              51
  0457  RRIGHT          02
  0083  RS              66
  0299  RTOLBYTES
  0394  SANY            01   /* same as FOREFF */
```

```
  0395 SAREG          02   /* same as INAREG */
* 0327 SAVEREGION     8 /* number of bytes for save area */
  0397 SBREG          010  /* same as INBREG */
  0399 SCC            040  /* same as FORCC */
* 0344 SCCON          (SPECIAL+100)
  0401 SCON           0200
  0116 SCONV          104
* 0102 SETBIT         90
  0225 SETOFF(x,y)    if( x%y != 0 ) x = ( (x/y + 1) * y)
  0532 SETSTO(x,y)    (stotree=(x),stocook=(y))
  0402 SFLD           0400
  0151 SHFFLG         010000
* 0035 SHIFTOP        29
  0166 SHORT          3
* 0346 SICON          (SPECIAL+101)
  0144 SIMPFLG        040
* 0056 SIZEOF         48
* 0073 SM             57
* 0410 SMONE          (SPECIAL|2)
  0400 SNAME          0100
* 0409 SONE           (SPECIAL|1)
  0403 SOREG          01000
* 0316 SP             6    /* stack pointer */
  0407 SPECIAL        0100000
  0153 SPFLG          040000
  0396 STAREG         04   /* same as INTAREG */
  0111 STARG          99
  0404 STARNM         02000
  0405 STARREG        04000
  0110 STASG          98
  0398 STBREG         020  /* same as INTBREG */
  0112 STCALL         100
* 0292 STDPRTREE
* 0284 STKREG         5
  0337 STOARG(p)      /* just evaluate the arguments,
  0339 STOFARG(p)
  0340 STOSTARG(p)
  0085 STREF          69
  0006 STRING         3
* 0038 STROP          32
  0171 STRTY          8
* 0043 STRUCT         35
  0406 SWADD          040000
* 0048 SWITCH         40
* 0215 SWITSZ         250 /* size of switch table */
* 0212 SYMTSZ         450 /* size of the symbol table */
  0250 SZCHAR         8
* 0253 SZDOUBLE       64
  0408 SZERO          SPECIAL
* 0252 SZFLOAT        32
  0251 SZINT          16
  0254 SZLONG         32
* 0256 SZPOINT        16
* 0255 SZSHORT        16
* 0429 TANY           010000  /* matches anything within reason */
  2451 TBUSY          01000
  0417 TCHAR          01
  0422 TDOUBLE        040
* 0103 TESTBIT        91
  0421 TFLOAT         020
  0419 TINT           04
  0420 TLONG          010
  0186 TMASK          060
* 0187 TMASK1         0300
* 0188 TMASK2         0360
  0335 TMPREG         R5
  0650 TNEXT(p)       (p== &node[TREESZ-1]?node:p+1)
* 0236 TNULL          PTR   /* pointer to UNDEF */
  0423 TPOINT         0100
  0428 TPTRTO         04000   /* pointer to one of the above */
  0218 TREESZ         350 /* space for building parse tree */
  0220 TREESZ         1000
  0191 TSHIFT         2
  0418 TSHORT         02
```

```
  0430  TSTRUCT         020000  /* structure or union */
  0424  TUCHAR          0200
  0427  TULONG          02000
  0426  TUNSIGNED       01000
  0425  TUSHORT         0400
  0141  TYFLG           016
  0041  TYPE            33
  0175  UCHAR           12
  0100  UGE             88
  0101  UGT             89
  0098  ULE             86
  0178  ULONG           15
  0099  ULT             87
  0129  UNARY           2+
* 0163  UNDEF           0
  0172  UNIONTY         9
* 0037  UNOP            31
* 0198  UNSIGNABLE(x)   ((x)<=LONG&&(x)>=CHAR)
  0177  UNSIGNED        14
  0232  UPKFOFF(v)      (v>>6)
  0231  UPKFSZ(v)       (v&077)
  0176  USHORT          13
  0137  UTYPE           04
* 0051  WHILE           43
  3118  ZCHAR           01
  3120  ZFLOAT          04
  3119  ZLONG           02
  0157  asgop(o)        (dope[o]&ASGFLG)
* 0582  callchk(x)      allchk(x)
  0159  callop(o)       (dope[o]&CALLFLG)
* 0341  genfcall(a,b)   gencall(a,b)
* 0526  isbreg(r)       (rstatus[r]&SBREG)
  0528  istnode(p)      (p->op==REG && istreg(p->rval))
  0527  istreg(r)       (rstatus[r]&(STBREG|STAREG))
* 0158  logop(o)        (dope[o]&LOGFLG)
* 0302  makecc(val,i)   lastcon = i ? (val<<8)|lastcon : val
  3115  max(x,y)        ((x)<(y)?(y):(x))
  3116  min(x,y)        ((x)<(y)?(x):(y))
* 0156  optype(o)       (dope[o]&TYFLG)
  0330  wdal(k)         (BYTEOFF(k)==0)
```

Appendix C. Procedure Calls Arranged by Caller

This table gives references to procedure calls (caller/callee) arranged alphabetically by caller. Recursion is denoted by an asterisk.

```
acon              4202
adrcon            4219
adrput            4224
|    acon         4233
|    |            4244
|    |            4249
|    |            4263
|    *            4271
|    *            4288
|    *            4294
|    cerror       4301
|    szty         4238
|    tfree        4296
|    tshape       4269
|    werror       4258
allchk            2479
|    cerror       2486
allo              2493
|    freereg      2502
|    |            2511
|    freetemp     2523
|    |            2527
allo0             2458
argsize           3668
|    *            3672
callreg           4021
canon             1307
|    ffld         1312
|    |            1313
|    fwalk        1313
|    oreg2        1309
|    |            1315
|    sucomp       1309
|    |            1319
|    walkf        1315
|    |            1319
cbgen             3981
|    cerror       3987
|    deflab       4005
|    expand       3996
|    |            4002
|    getlab       3995
|    reclaim      4006
cbranch           1806
|    cbgen        1852
|    |            1857
|    |            1868
|    |            1915
|    |            1916
|    *            1874
|    *            1875
|    *            1882
|    *            1883
|    *            1889
|    *            1896
|    *            1902
|    *            1903
|    *            1905
|    codgen       1849
|    |            1851
|    |            1866
|    |            1894
|    |            1914
|    deflab       1876
|    |            1884
|    |            1904
|    |            1906
|    |            1907
|    getlab       1873
|    |            1881
|    |            1900
|    |            1901
|    |            1902
|    reclaim      1869
|    |            1917
cerror            0621
|    where        0622
codgen            1281
|    canon        1289
|    eprint       1293
|    fwalk        1293
|    order        1300
|    |            1302
|    store        1295
conput            4309
|    acon         4313
|    cerror       4321
constore          1451
|    *            1464
|    markcall     1462
|    store        1468
deflab            3358
delay             1183
|    codgen       1196
|    |            1198
|    delay1       1191
|    delay2       1195
delay1            1202
|    delay        1219
|    *            1208
|    *            1216
|    *            1229
|    ncopy        1223
delay2            1233
|    *            1275
|    *            1276
|    deltest      1261
|    ncopy        1267
|    tcopy        1264
deltest           2947
|    spsz         2950
eobl2             3755
eprint            1134
|    adrput       1154
|    tprint       1167
eread             1089
|    cerror       1113
|    *            1127
|    *            1128
|    rbusy        1118
|    rdin         1098
|    |            1103
|    |            1104
|    |            1106
|    |            1111
|    |            1112
|    talloc       1099
expand            2376
|    adrcon       2410
|    adrput       2436
|    conput       2428
|    getlr        2422
|    |            2428
|    |            2432
|    |            2436
|    |            2440
|    hopcode      2418
|    insput       2432
|    upput        2440
|    zzzcode      2389
ffld              1928
|    rewfld       1939
|    szty         1941
|    |            1947
|    talloc       1955
|    |            1968
|    |            1973
flshape           4195
|    shumul       4198
freereg           2546
|    callreg      2555
|    usable       2556
|    |            2560
|    |            2565
|    |            2572
freetemp          2647
fwalk             0699
|    *            0710
genargs           3623
|    canon        3645
|    |            3648
|    cerror       3649
|    expand       3657
|    *            3629
|    offstar      3647
|    order        3664
|    reclaim      3659
gencall           4032
|    argsize      4037
|    genargs      4041
|    match        4049
|    order        4045
|    popargs      4050
|    shltype      4044
genscall          4026
|    gencall      4028
getlab            3353
getlr             2214
|    cerror       2233
hardops           3802
|    cerror       3859
|    talloc       3839
|    |            3844
|    |            3866
|    |            3876
hopcode           4399
|    cerror       4411
insput            4326
|    cerror       4327
lastchan          4085
lineid            3770
main              0961
|    allchk       1038
|    cerror       0995
|    |            1014
|    |            1043
|    delay        1035
|    eobl2        1012
|    eprint       1028
|    eread        1026
|    fwalk        1028
|    lineid       1022
|    p2init       0968
|    rdin         0990
|    |            0992
|    |            0993
|    |            1019
|    reclaim      1036
|    setregs      1007
|    tcheck       1039
|    tinit        0969
markcall          1420
|    *            1440
match             2159
|    allo         2202
|    expand       2205
|    getlr        2191
|    |            2194
|    rcount       2168
|    reclaim      2206
|    shltype      2184
|    tshape       2192
|    |            2195
|    ttype        2193
|    |            2196
mkadrs            2968
|    cerror       2981
mkdope            0811
mkrall            3093
|    rallo        3111
myreader          3926
|    canon        3928
|    hardops      3927
|    optim2       3929
|    walkf        3927
|    |            3929
ncopy             2891
nextcook          4075
niceuty           3604
|    shumul       3609
notoff            3613
offstar           3363
|    order        3370
|    |            3374
optim2            3888
|    talloc       3913
order             1524
|    canon        1538
|    |            1744
|    cbgen        1635
|    cbranch      1622
|    |            1628
|    |            1651
|    cerror       1604
```

```
|    odgen       1607
|    |           1609
|    |           1630
|    |           1637
|    deflab      1636
|    |           1638
|    eprint      1545
|    |           1747
|    fwalk       1545
|    |           1747
|    gencall     1688
|    genscall    1695
|    getlab      1628
|    |           1635
|    |           1651
|    lastchan    1796
|    match       1560
|    |           1793
|    ncopy       1610
|    nextcook    1563
|    offstar     1659
|    |           1709
|    *           1654
|    *           1664
|    *           1670
|    *           1705
|    *           1749
|    *           1750
|    prcook      1543
|    |           1596
|    |           1598
|    rallo       1539
|    |           1745
|    rcount      1537
|    reclaim     1634
|    |           1725
|    |           1742
|    |           1785
|    setasg      1754
|    setasop     1736
|    setbin      1759
|    setincr     1713
|    setstr      1732
|    tcopy       1724
|    |           1740
|    tfree       1643
|    tshape      1791
|    uerror      1674
preg2            1088
|    notoff      2065
|    szty        2017
|    |           2026
|    |           2054
|    tfree       2071
p2init           0890
|    allo0       0897
|    cerror      0949
|    mkdope      0955
|    setrew      0956
popargs          4055
prcook           1492
rallo            3006
|    mkrall      3043
|    |           3049
|    |           3064
|    *           3045
|    *           3056
|    *           3059
|    *           3087
|    *           3088
rbusy            2874
|    cerror      2886
|    szty        2879
|    |           2883
rcount           1516
|    cerror      1518
rdin             1055
|    cerror      1064
|    |           1077
|    |           1082
recl2            2839
|    rfree       2841
|    |           2844
|    |           2845
|    |           2848
reclaim          2677
|    cerror      2732
|    |           2748
|    |           2782
|    |           2797
|    |           2801
|    prcook      2690
|    rbusy       2784
|    recl2       2697
|    rfree       2785
|    rmove       2786
|    rwprint     2688
|    szty        2781
|    tcopy       2758
|    tfree       2706
|    |           2714
|    |           2759
|    tshape      2742
|    walkf       2697
rewfld           4091
rfree            2854
|    cerror      2863
|    |           2866
|    |           2868
|    szty        2859
|    |           2864
rmove            4378
rwprint          2806
setasg           3492
|    offstar     3497
|    |           3509
|    |           3513
|    order       3499
|    |           3504
|    |           3518
|    tshape      3508
setasop          3398
|    canon       3472
|    cerror      3462
|    |           3465
|    offstar     3438
|    |           3442
|    |           3481
|    order       3414
|    |           3424
|    |           3474
|    |           3475
|    |           3487
|    rallo       3473
|    reclaim     3454
|    shumul      3448
|    tcopy       3452
setbin           3525
|    cerror      3578
|    niceuty     3553
|    |           3562
|    |           3566
|    |           3584
|    offstar     3535
|    |           3554
|    |           3563
|    |           3567
|    |           3585
|    order       3536
|    |           3540
|    |           3558
|    |           3571
|    |           3575
|    |           3590
|    |           3595
|    shumul      3534
setincr          3378
setregs          3739
setrew           2112
|    cerror      2123
|    shltype     2147
setstr           3383
|    cerror      3390
|    order       3385
|    |           3391
shareit          2620
|    ushare      2623
|    |           2624
shltype          4141
|    shumul      4143
shtemp           4187
shumul           4147
|    spsz        4156
special          4163
|    cerror      4179
spsz             4096
stoarg           1392
|    *           1396
|    store       1409
|    |           1414
stoasg           2960
|    shltype     2964
store            1325
|    constore    1370
|    markcall    1357
|    |           1365
|    mkadrs      1383
|    stoarg      1352
|    stoasg      1345
|    |           1380
|    *           1351
|    *           1359
|    *           1376
|    *           1387
|    *           1388
sucomp           3122
|    shumul      3146
|    szty        3131
|    |           3194
|    zum         3195
|    |           3196
|    |           3225
|    |           3305
szty             4126
talloc           0653
|    cerror      0660
tcheck           0665
|    cerror      0670
|    tinit       0671
tcopy            2910
|    ncopy       2916
|    rbusy       2919
|    |           2922
|    |           2923
|    |           2926
|    talloc      2916
|    *           2933
|    *           2935
tfree            0675
|    tfree1      0676
|    |           0678
|    walkf       0678
tfree1           0682
|    cerror      0683
tinit            0642
tprint           0821
tshape           2238
|    flshape     2284
|    shtemp      2268
|    shumul      2317
|    special     2262
ttype            2325
|    *           2339
uerror           0599
|    cerror      0607
|    where       0604
upput            4331
|    acon        4345
|    |           4352
|    |           4364
|    cerror      4369
|    werror      4362
usable           2582
|    cerror      2586
|    shareit     2601
|    |           2602
|    |           2615
|    szty        2595
ushare           2629
|    getlr       2632
|    szty        2641
walkf            0688
|    *           0693
|    *           0694
werror           0612
|    where       0613
where            3776
zum              3318
zzzcode          4415
|    adrput      4603
|    cbgen       4425
|    |           4436
|    cerror      4454
|    |           4455
|    |           4595
|    |           4606
|    |           4612
|    |           4628
|    |           4632
|    |           4660
|    conput      4556
|    |           4563
|    deflab      4426
|    |           4431
|    expand      4509
|    |           4510
|    |           4642
|    |           4645
|    getlab      4425
|    getlr       4427
|    |           4430
|    |           4496
|    |           4556
|    |           4563
|    |           4603
|    rbusy       4592
|    rfree       4589
|    tcopy       4444
```

Appendix D. Procedure Calls Arranged by Callee

This table gives references to procedure calls (caller/callee) arranged alphabetically by callee.

```
              acon 4202
adrput          |  4233
  |             |  4244
  |             |  4249
  |             |  4263
conput          |  4313
upput           |  4345
  |             |  4352
  |             |  4364
            adrcon 4219
expand          |  2410
            adrput 4224
eprint          |  1154
expand          |  2436
zzzcode         |  4603
            allchk 2479
main            |  1038
              allo 2493
match           |  2202
             allo0 2458
p2init          |  0897
           argsize 3668
gencall         |  4037
           callreg 4021
freereg         |  2555
             canon 1307
codgen          |  1289
genargs         |  3645
  |             |  3648
myreader        |  3928
order           |  1538
  |             |  1744
setasop         |  3472
             cbgen 3981
cbranch         |  1852
  |             |  1857
  |             |  1868
  |             |  1915
  |             |  1916
order           |  1635
zzzcode         |  4425
  |             |  4436
           cbranch 1806
order           |  1622
  |             |  1628
  |             |  1651
            cerror 0621
adrput          |  4301
allchk          |  2486
cbgen           |  3987
conput          |  4321
eread           |  1113
genargs         |  3649
getlr           |  2233
hardops         |  3859
hopcode         |  4411
insput          |  4327
main            |  0995
  |             |  1014
  |             |  1043
```

```
mkadrs          |  2981
order           |  1604
p2init          |  0949
rbusy           |  2886
rcount          |  1518
rdin            |  1064
  |             |  1077
  |             |  1082
reclaim         |  2732
  |             |  2748
  |             |  2782
  |             |  2797
  |             |  2801
rfree           |  2863
  |             |  2866
  |             |  2868
setasop         |  3462
  |             |  3465
setbin          |  3578
setrew          |  2123
setstr          |  3390
special         |  4179
talloc          |  0660
tcheck          |  0670
tfree1          |  0683
uerror          |  0607
upput           |  4369
usable          |  2586
zzzcode         |  4454
  |             |  4455
  |             |  4595
  |             |  4606
  |             |  4612
  |             |  4628
  |             |  4632
  |             |  4660
            codgen 1281
cbranch         |  1849
  |             |  1851
  |             |  1866
  |             |  1894
  |             |  1914
delay           |  1196
  |             |  1198
order           |  1607
  |             |  1609
  |             |  1630
  |             |  1637
            conput 4309
expand          |  2428
zzzcode         |  4556
  |             |  4563
          constore 1451
store           |  1370
            deflab 3358
cbgen           |  4005
cbranch         |  1876
  |             |  1884
  |             |  1904
  |             |  1906
```

```
  |             |  1907
order           |  1636
  |             |  1638
zzzcode         |  4426
  |             |  4431
             delay 1183
delay1          |  1219
main            |  1035
            delay1 1202
delay           |  1191
            delay2 1233
delay           |  1195
           deltest 2947
delay2          |  1261
             eobl2 3755
main            |  1012
            eprint 1134
codgen          |  1293
main            |  1028
order           |  1545
  |             |  1747
             eread 1089
main            |  1026
            expand 2376
cbgen           |  3996
  |             |  4002
genargs         |  3657
match           |  2205
zzzcode         |  4509
  |             |  4510
  |             |  4642
  |             |  4645
              ffld 1928
canon           |  1312
  |             |  1313
           flshape 4195
tshape          |  2284
           freereg 2546
allo            |  2502
  |             |  2511
          freetemp 2647
allo            |  2523
  |             |  2527
             fwalk 0699
canon           |  1313
codgen          |  1293
main            |  1028
order           |  1545
  |             |  1747
           genargs 3623
gencall         |  4041
           gencall 4032
genscall        |  4028
order           |  1688
          genscall 4026
order           |  1695
            getlab 3353
cbgen           |  3995
cbranch         |  1873
  |             |  1881
```

```
  |             |  1900
  |             |  1901
  |             |  1902
order           |  1628
  |             |  1635
  |             |  1651
zzzcode         |  4425
             getlr 2214
expand          |  2422
  |             |  2428
  |             |  2432
  |             |  2436
  |             |  2440
match           |  2191
  |             |  2194
ushare          |  2632
zzzcode         |  4427
  |             |  4430
  |             |  4496
  |             |  4556
  |             |  4563
  |             |  4603
           hardops 3802
myreader        |  3927
           hopcode 4399
expand          |  2418
            insput 4326
expand          |  2432
          lastchan 4085
order           |  1796
            lineid 3770
main            |  1022
              main 0961
          markcall 1420
constore        |  1462
store           |  1357
  |             |  1365
             match 2159
gencall         |  4049
order           |  1560
  |             |  1793
            mkadrs 2968
store           |  1383
            mkdope 0811
p2init          |  0955
            mkrall 3093
rallo           |  3043
  |             |  3049
  |             |  3064
          myreader 3926
             ncopy 2891
delay1          |  1223
delay2          |  1267
order           |  1610
tcopy           |  2916
          nextcook 4075
order           |  1563
           niceuty 3604
setbin          |  3553
  |             |  3562
```

¦		3566
¦		3584
	notoff	3613
oreg2		2065
	offstar	3363
genargs		3647
order		1659
¦		1709
setasg		3497
¦		3509
¦		3513
setasop		3438
¦		3442
¦		3481
setbin		3535
¦		3554
¦		3563
¦		3567
¦		3585
	optim2	3888
myreader		3929
	order	1524
codgen		1300
¦		1302
genargs		3664
gencall		4045
offstar		3370
¦		3374
setasg		3499
¦		3504
¦		3518
setasop		3414
¦		3424
¦		3474
¦		3475
¦		3487
setbin		3536
¦		3540
¦		3558
¦		3571
¦		3575
¦		3590
¦		3595
setstr		3385
¦		3391
	oreg2	1988
canon		1309
¦		1315
	p2init	0890
main		0968
	popargs	4055
gencall		4050
	prcook	1492
order		1543
¦		1596
¦		1598
reclaim		2690
	rallo	3006
mkrall		3111
order		1539
¦		1745
setasop		3473
	rbusy	2874
eread		1118
reclaim		2784
tcopy		2919
¦		2922
¦		2923
¦		2926
zzzcode		4592
	rcount	1516
match		2168
order		1537
	rdin	1055
eread		1098
¦		1103
¦		1104
¦		1106
¦		1111
¦		1112
main		0990
¦		0992
¦		0993
¦		1019
	recl2	2839
reclaim		2697
	reclaim	2677
cbgen		4006
cbranch		1869
¦		1917
genargs		3659
main		1036
match		2206
order		1634
¦		1725
¦		1742
¦		1785
setasop		3454
	rewfld	4091
ffld		1939
	rfree	2854
recl2		2841
¦		2844
¦		2845
¦		2848
reclaim		2785
zzzcode		4589
	rmove	4378
reclaim		2786
	rwprint	2806
reclaim		2688
	setasg	3432
order		1754
	setasop	3398
order		1736
	setbin	3525
order		1759
	setincr	3378
order		1713
	setregs	3739
main		1007
	setrew	2112
p2init		0956
	setstr	3383
order		1732
	shareit	2620
usable		2601
¦		2602
¦		2615
	shltype	4141
gencall		4044
match		2184
setrew		2147
stoasg		2964
	shtemp	4187
tshape		2268
	shumul	4147
flshape		4198
niceuty		3609
setasop		3448
setbin		3534
shltype		4143
sucomp		3146
tshape		2317
	special	4163
tshape		2262
	spsz	4096
deltest		2950
shumul		4156
	stoarg	1392
store		1352
	stoasg	2960
store		1345
¦		1380
	store	1325
codgen		1295
constore		1468
stoarg		1409
¦		1414
	sucomp	3122
canon		1309
¦		1319
	szty	4126
adrput		4238
ffld		1941
¦		1947
oreg2		2017
¦		2026
¦		2054
rbusy		2879
¦		2883
reclaim		2781
rfree		2859
¦		2864
sucomp		3131
¦		3194
usable		2595
ushare		2641
	talloc	0653
eread		1099
ffld		1955
¦		1968
¦		1973
hardops		3839
¦		3844
¦		3866
¦		3876
optim2		3913
tcopy		2916
	tcheck	0665
main		1039
	tcopy	2910
delay2		1264
order		1724
¦		1740
reclaim		2758
setasop		3452
zzzcode		4444
	tfree	0675
adrput		4296
order		1643
oreg2		2071
reclaim		2706
¦		2714
¦		2759
	tfree1	0682
tfree		0676
¦		0678
	tinit	0642
main		0969
tcheck		0671
	tprint	0821
eprint		1167
	tshape	2238
adrput		4269
match		2192
¦		2195
order		1791
reclaim		2742
setasg		3508
	ttype	2325
match		2193
¦		2196
	uerror	0599
order		1674
	upput	4331
expand		2440
	usable	2582
freereg		2556
¦		2560
¦		2565
¦		2572
	ushare	2629
shareit		2623
¦		2624
	walkf	0688
canon		1315
¦		1319
myreader		3927
¦		3929
reclaim		2697
tfree		0678
	werror	0612
adrput		4258
upput		4362
	where	3776
cerror		0622
uerror		0604
werror		0613
	zum	3318
sucomp		3195
¦		3196
¦		3225
¦		3305
	zzzcode	4415
expand		2389