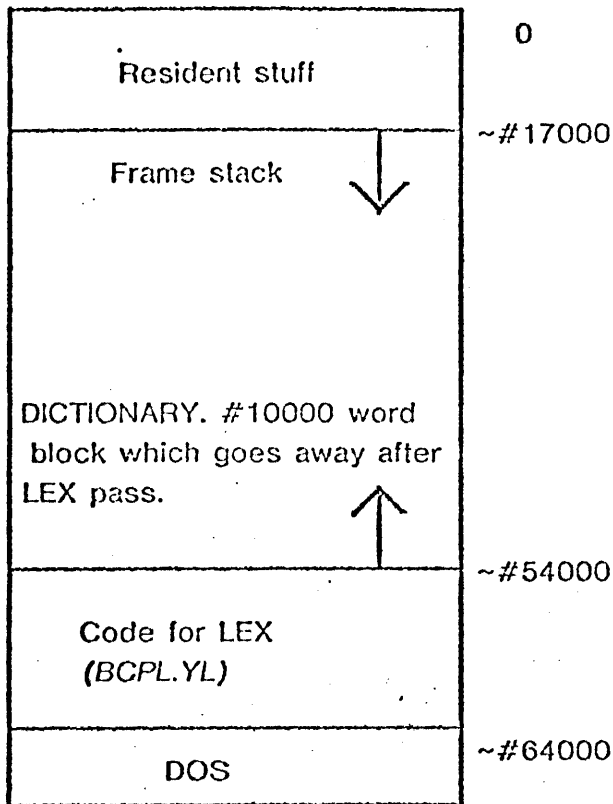|	
page 0 stuff	0
page 1 stuff (DOS)	#400
two machine language routines (from BCPLD/1 in load string)	#1000
Nova debugger (may not be present)	~#1000
BCPL runtime routines	~#4400
statics	~#5300
BCPL0 *BCPL1* *BCPL2*	~#6200
BCPL stack frames ↓	~#17000

BCPL stack frames grow from here towards DOS.

If these two areas collide, the compiler aborts.

Free storage grows from beginning of overlay area towards 0. Beginning of free storage depends upon size of overlay code.
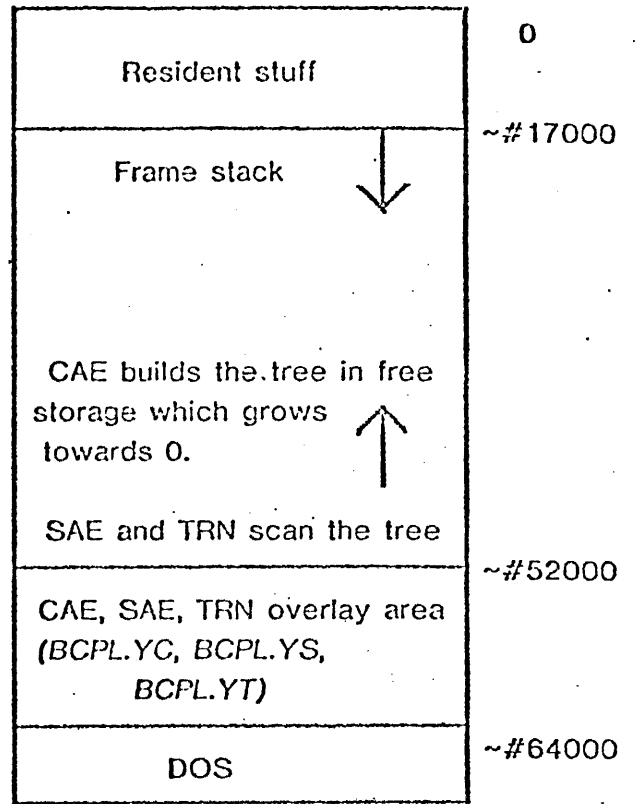
↑ Free storage	
Compiler overlay area	
DOS	~#64000

COMPILER CORE ORGANIZATION

LEX Core Organization

Resident stuff
Frame stack ↓
DICTIONARY. #10000 word block which goes away after LEX pass. ↑
Code for LEX (BCPL.YL)
DOS

0
~#17000
~#54000
~#64000

The length of LEX code is remembered with A/X during loading.

CAE, SAE, TRN Core Organization

Resident stuff
Frame stack ↓
CAE builds the tree in free storage which grows towards 0. ↑ SAE and TRN scan the tree
CAE, SAE, TRN overlay area (BCPL.YC, BCPL.YS, BCPL.YT)
DOS

0
~#17000
~#52000
~#64000

The Max length(CAE, SAE, TRN) is remembered by B/X during loading.

NCG Core Organization

Resident stuff
Frame stack ↓
Free storage for dictionary and tables ↑
#5000 word code block
Code for NCG (BCPL.YG)
DOS

0
~#17000
~#33000
~#40000
~#64000
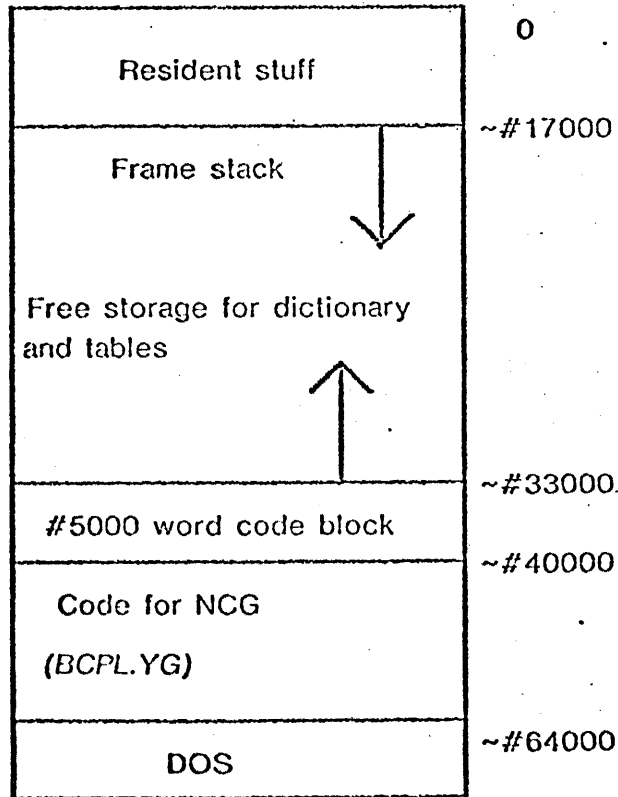
The length of NCG code is remembered with C/X during loading.

COMPILER CORE ORGANIZATION (2)

FOR
ptr to exp
ptr to exp
0 or ptr to exp
ptr to statement

CASETO
ptr to exp
ptr to exp
ptr to statement
(character number)

DEFAULT
ptr to statement

SWITCHON
ptr to exp
ptr to statement

REPEAT
ptr to statement

(LOCAL) + name
0

CASE
ptr to exp
ptr to statement
(character number)

REPEATWHILE/REPEATUNTIL
ptr to exp
ptr to statement

Return/Endcase/Loop/Break/Finish/Abort

(One word statements are one word nodes)

Control Nodes

(figure 5)

FNAP
ptr to exp
0 or ptr to exp

COND
ptr to exp
ptr to exp
ptr to exp

SIZE/OFFSET
0
Qualdescriptor

LEFTLUMP/RIGHTLUMP
ptr to exp
0
Qualdescriptor

(Binary Operator)
ptr to exp
ptr to exp

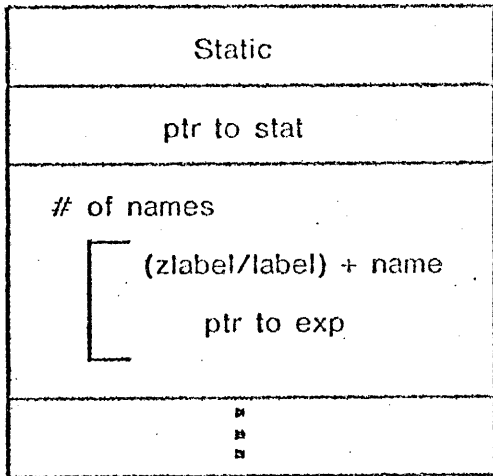(VECAP MULT DIF REM LSHIFT RSHIFT PLUS MINUS EQ NE GR GE LS LE LOGAND LOGOR EQV NEQV)

Qualdescriptor:

Names
#100000+name1
subscripts on name1
ptr to exp11
$\vdots$
ptr to exp 1n
$\vdots$
#100000+nameK
subscripts on nameK
ptr to exp1,K
$\vdots$
ptr to expN,K

More Expressnion Nodes
(Figure 7)

Line
(char number)
ptr to stat

Structure
ptr to structdef
ptr to stat

Static
ptr to stat
# of names
(zlabel/label) + name
ptr to exp
¤¤¤

Let
ptr to def
ptr to stat

ext
ptr to stat
# of names
(extlabel/zextlabel) + name
0
¤¤¤

Manifest
ptr to stat
# of names
(constant) + name
ptr to exp
¤¤¤

STATEMENT NODES

(FIGURE 1)

DEFNODES

(Figure 3)

Def:

AND
ptr to line
ptr to line

VALDEF
ptr to namelist
ptr to explist
0

FNDEF
0 or ptr to namelist
ptr to exp
0
0
0   or

LINE
(character number)
ptr to and/def

(LABEL) + name
0

(LOCAL) + name
0

RTDEF
0 or ptr to namelist
ptr to statement
0
0
or   0

Explist:

COMMA
ptr to exp
ptr to explist

Namelist:

(LOCAL) + name
0

NIL
0

COMMA
ptr to namenode/nil
ptr to namelist

Expressions:

(Note: Anywhere in the tree, "ptr to exp" is only a pointer to one of the following nodes if bit0 is 0. If a word with bit0 = 1 appears where a "ptr to exp" is expected, the right hand 12 bits are the dictionary identifier for a name. This is done to minimize tree size.)

NUMBER
value

TABLE
length
ptr to exp1
. . .
ptr to expn

STRINGCONSTANT
length   char 1
. . .

CHARCONSTANT
charcode

NIL/TRUE/FALSE
0/-1/0

VAOLF
ptr to statement

LV/RV/NEG/NOT/VEC
ptr to exp.
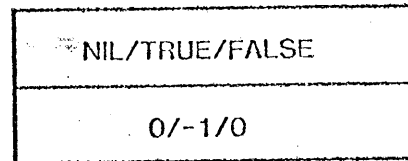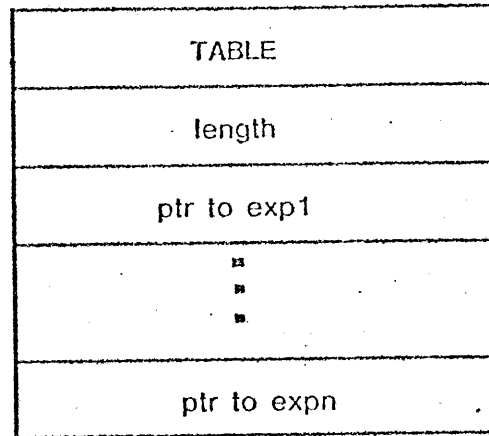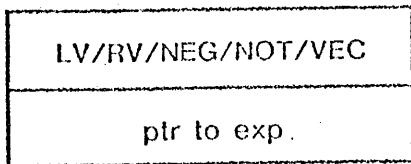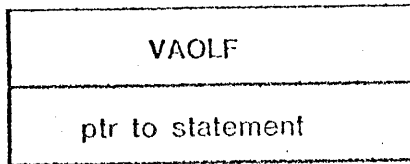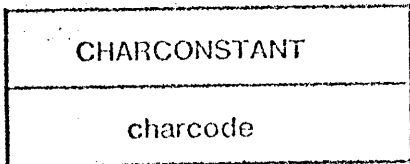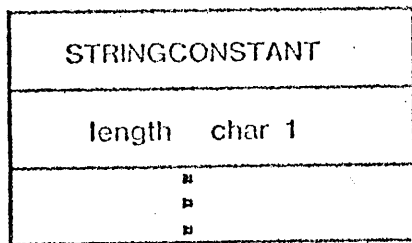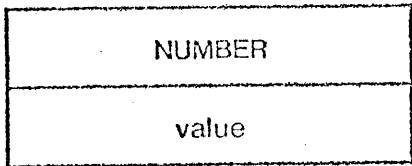
Expression Nodes

(Figure 6)

STRUCTDEFS:

FIELDLIST
# of fields
ptr to struct def 1
⁞
ptr to struct def n

RV
#100000 + name

BLANK
ptr to structdef/bitdef

OVERLAYLIST
# of overlays
ptr to struct def 1
⁞
ptr to struct def n

#100000 + NAME
ptr to structdef/bitdef

BITDEFS:

BIT
0 or ptr to exp

UPLUMP
ptr to struct def
0 or ptr to exp
ptr to exp

BYTE
0/ptr to exp

WORD
0 or ptr to exp

STRUCTURE NODES

(Figure 2)

Statements:

SEQ
ptr to statement
ptr to statement

IF/UNLESS/WHILE/UNTIL
ptr to exp
ptr to statement

COLON
ptr to namelist
ptr to statement
0
0

TEST	
ptr to exp	
ptr to statement	(true)
ptr to statement	(false)

ASS
ptr to namelist
ptr to exp

GOTO
ptr to exp

RESULTS
ptr to exp

RTAP
ptr to exp
0 or ptr to explist

(LABEL) + NAME
0
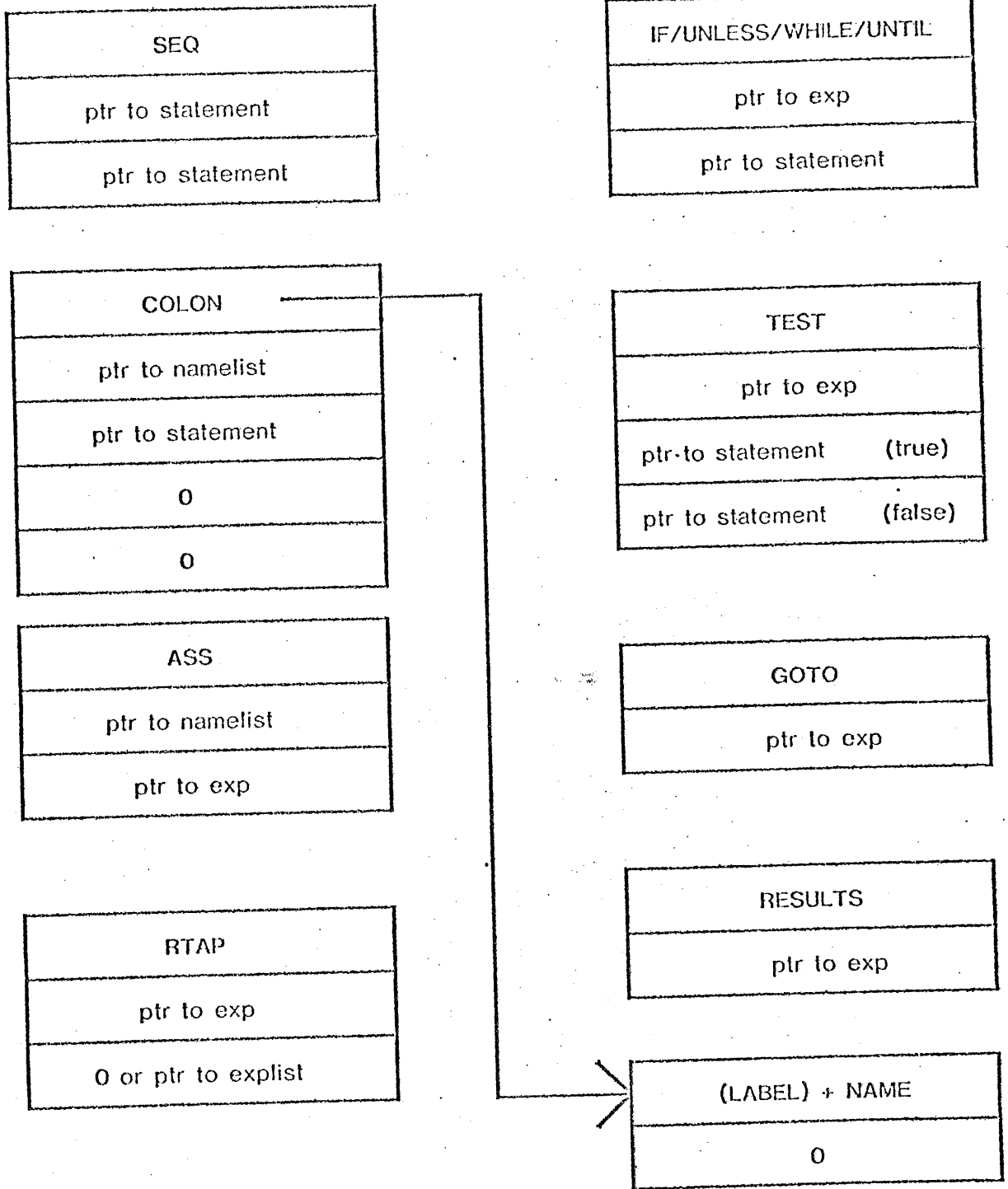
Statement Nodes

(Figure 4)

Font Conventions

Italics represents file names: *BCPL0*.
Bold represents procedures, manifests or statics: **ReadSym()**, **Ch**
Bold Italics represent lexemes, tree nodes, and ocode: ***AND***, ***VALDEF***.
Note: Font conventions do not apply to headings.

Compiler Core Organization

Use pictures!!

0...777

    standard page 0, 1 stuff

1000

    two machine language routines
    (from *BCPLD/1* in load string)

...~1000

    Nova debugger
    (may not be present)

...~4400

    standard *BCPL* non-time routines

...~5300

    statics

...~6200

    BCPL0 (main program)
    BCPL1 (I/O procedures)
    BCPL2 (other utility procedures)

...~17000

    The *BCPL* frame stack grows
    from here towards DOS.

        If frame space and free space collide (during a
        procedure call, or a free storage block is allocated)
        the compiler aborts.

    Free storage space   for each pass
    grows towards 0 from the beginning
    of the code for the pass

        The start of free storage for a given
        pass depends on the size of its code.

    Code for the individual passes
    is read in just above DOS.

...64000

    DOS

...77777

LEX Core Organization

0

      Resident stuff

...~17000

      Frame stack

...~44000

      Dictionary (#10,000 word free storage block
      which goes away after LEX pass)

*...~54000

      Code for LEX (overlay file *BCPL.YL*)

...~64000

      DOS

...77777

* The start of free space for LEX is (top of DOS) - (length of LEX
  code). The length of the LEX code is remembered during loading with
  A/X.

CAE, SAE, TRN Core Organization

0...*17000

   Resident stuff

...~17000

   frame stack

   CAE builds the tree in free storage, which grows
   from ~52000 towards 0. SAE and TRN scan the tree.

*...~52000

   Code for each of the CAE, SAE, AND TRN passes
   (*BCPL.YC, BCPL.YS, BCPL.YT* overlay files) is read
   in here at the beginning of the pass.

...~64000

   DOS

...77777

*The start of free space is (top of DOS) - max(length of CAE/SAE/TRN code). The
maximum length is remembered during compiler loading with B/X.

NCG Core Organization

0

        Resident Stuff

...~17000

        Frame Stack

        NCG alocates space for the dictionary, and for various
        tables, above the fixed-length code block.

...~33000

        Code (#5000 word free-storage block)

*...~400000

        Code for NCG (overlay file *BCPL.YG*)

...~64000

        DOS

...~77777

* The start of free storage for NCG is (top of DOS) - (length of NCG code). The
length of NCG code is remembered during loading with C/X.

LEX

LEX is the lexical analysis pass. It reads the source file (and any of its "get" files) and converts the source stream into a stream of lexemes. The lexeme stream is written as it is generated onto the temporary file *$$$.BL* (which will be read by CAE). LEX builds an identifier dictionary in core, and writes the dictionary onto the temporary file *$$$.BD* when it is done. (The dictionary is used by NCG to output external name strings for the loader; SAE also uses the dictionary to print undefined name error messages.)

The lexeme stream is essentially a stream of bytes, one byte for each lexical unit recognized in the source text; a few lexemes (*NAMEBRA, NAMEKET, LINE NUMBER, NAME, STRINGCONST, CHARCONST*) are followed in the lexeme stream by additional bytes of information. The byte values for the lexemes are defined as manifests in the "get" file *BLEXX*; many of these definitions have bits set in the left half, used by the procedure LexOut to do automatic semicolon insertion. These lexeme definitions are duplicated in *BCAEX* (without the left-half bits), since CAE reads the lexeme stream. Many lexemes are also used as tree node identifiers, and therefore are declared in *BSAEX* and *BTRNX*; some are also used as ocode operators, and appear in *BNCGX* as well.

The driver for LEX is ReadSource in *BLEX0*; it is called by the compiler's main program after the LEX overlay file (*BCPL.YL*) is read in and the free storage pointer is initialized.

ReadSource()

1. Creates space for a few vectors used by LEX.

2. Allocates a free storage block for the Dictionary.
   (see below for dictionary structure)

3. Opens the source file.

4. Opens the temp file *$$$.BL*
   (See the description of OpenTemp, ResetStream, etc.)

5. Main Loop: calls Readsymb repeatedly, until the end of the source text is reached.

6. Cleans up the source file line pointer table.

7. Writes out the dictionary onto the temp file *$$$.BD*.

The main procedure of the LEX phase is Readsymb in *BLEX1*. Each time it is called, it outputs a lexeme to the file *$$$.BL* (via LexOut(lexeme)) and returns. LexOut handles the automatic insertion of semicolons and *DO's*; when this happens, the flag ReadAhead is set, so that the next call on Readsymb will simply output the lexeme that caused the semicolon or *DO* to be inserted. LexOut also outputs extra information for the lexemes which require it.

Normal entry to Readsymb() reads a char into Ch (via Rch() in *BLEX0*), and branches on the character class; as determined by Kind(Ch):

Ignorable (space, tab): ignore
Digit: start of a decimal number; read it and output *NUMBER* lexeme.
Capital, Small (alphabetic): start of a reserved word or identifier
Default: either a special char ("+", ">", etc) or illegal; branch on the character itself.

If a name is read (string of letters & digits), it is looked up by ReservedWord() (in *BLEX2*). If it is a reserved word (other than "get") the corresponding lexeme is output. If it is "get", the file name is read and the file is opened. Otherwise, the name is an identifier, and is entered in the dictionary; the lexeme *NAME* is output.

If a special char is read, some code appropriate to the character is executed. For most characters, this is just LexOut(appropriate lexeme), perhaps reading ahead one character, as for ">". Some characters, like "#" and brackets, require some extra work.

Source File Handling

Throughout compilation, pointers to the source text are needed for printing error messages. A source text pointer must fit in one word, since there is at least one node in the parse tree for every source line and every statement in the program; so sequential character number is used. This means that there must be a way of converting a sequential character number to an output file name and a character number within the input file. (Remember that "get" file input is intermixed with main source file input, and that "get" files may be nested.) LEX constructs the following tables for this purpose:

GetnameV   (two words per source file, indexed by file number)

GetlineV   (three words per entry)

GetnameV: ptr to name of file1, len of file 1
            ptr to name of file 2, len of file 2,
            etc.

GetnameP   contains the number of source files *2.

An entry is made in GetlineV each time a "get" file is opened or closed. An entry contains:

```
----------------------------------------------
sequential char number of last char read
from the previous source
----------------------------------------------
previous source file number
----------------------------------------------
corresponding char number in the source
file
----------------------------------------------
```

So once GetlineV is complete, it can be used to convert a sequential char number to a source file number and a character number in that source file. One must simply find the first GetlineV entry whose first word is larger than the sequential character number, and offset the sequential char number by the difference between the first and third words of that entry.

This is what the procedure WriteLine (in *BCPL0*) does.

To Add a Reserved Word to *BCPL:*

Choose a lexeme name and a byte value.

The value must be one which is not used for any existing lexeme (*BLEXX*), tree node id (*BCAEX*) or ocode opcode (*BTRNX*).

Define the name in *BLEXX* with bits b1, b2, e1, e2, as follows:

If the reserved word must start a statement (like "switchon", "goto"), set "b2".

If the word might start a statement (like "rv" or "("), or if it begins a declaration (like "static"), set "b1".

If the word might end a statement (like "]" or ")") set "e1".

See the existing definitions for more examples of where the bits are used.

Add the new reserved word to the ReservedWord procedure in *BLEX2*.

Under "case $<initial letter>", add
" (veq("<rest of word>")?<lexeme>, "

and add an extra parenthesis to the "0)))..." at the end of the case expression.

Now LEX will output the new lexeme whenever it encounters the new reserved word in the source text.

Of course, you must define the lexeme in *BCAEX*, and add the appropriate code to CAE to parse and build the tree segment for the new statement or expression.

To Add a New Special Character or Character Combinations:

Choose a lexeme name and byte value, and declare it as for a new reserved word.

In Readsymb in *BLEX?*, there is a long "switchon" statement whose "cases" are character constants. A "case" must be added for the new character, or, if one already exists, it must be modified. Use "case$=" for a model of how to read ahead to check for a character combination.

The unused characters are currently:

←	(left arrow)
'	(single quote)
	(reverse single quote)
\	(back slash)
{,}	(curly brackets)
~ (¬) (∧)	(tilda, not, hat)
I	(vertical bar)

Dictionary

The first word of the dictionary is the current length of the dictionary.

The next 52_{10} words of the dictionary are the heads of sorted lists of dictionary entries, one for ea
upper- and lower-case letter, in the order a, A, b, B, . . . ,z, Z. Each list looks like:

```
         ----------        -----------       ----------        ----------
head     ptr to 1st 'a'    ptr to  1st 'A'   ptr to 1st 'b'        0
         ----------        -----------       ----------        ----------
              a:           n    c1      A:   BCPL              BCPL
                           c2   c3           string   . . . . string
                           (BCPL
                            string)
                           ----------        ---------        ----------
```

The pointers are relative to the first word of the dictionary. The list is sorted algebraically on the
word of the string, then on the second, etc. (This is not quite alphabetical order, because the first
has the length of the string in its left half.)

Most Important Routines in CAE: Readblockbody, Rcom, Rexp

There are two main classes of tree nodes: "exp" and "stat", corresponding to expressions and statements.

"stat" nodes are constructed by Readblockbody()*BCAE1* and by Rcom(Nextfn) (*BCAE3*).

Readblockbody() is used where a declaration is legal as the next statement. It calls Rcom(Readblockbody) if the next statement is not a declaration.

Rcom(Rcom) is used where a declaration is not permitted (that is, where a declaration, if it appears, must be enclosed in brackets, as after "if exp then . . .").

Rcom(Nextfn) calls Nextfor(Nextfn) after processing a statement which does not affect the class of statement to be expected. So Rcom(Rcom) will parse a sequence of non-declaration statements; Rcom(Readblockbody) will parse a sequence of statements which includes declarations [Readblockbody calls Rcom(Readblockbody) when a non-declaration statement is seen.]

"exp" nodes are constructed by Rexp(n), where "n" is an integer corresponding to the precedence of the expression to be passed.

Names appear in the tree in two different contexts:

1. In a node which

> *STATIC*
> *EXT*
> *MANIFEST*
> *VALDEF*
> *FNDEF*
> *RTDEF*
> *COLON*
> *FOR*

2. Wherever "---exp" can appear in a tree node.
   (That is, a use of the name as an expression.)

1. Wherever a name is declared, two words are allocated for it; either in the declaration node itself (*STATIC, EXT, MANIFEST*) or as a separate block pointed at by the declaration node (*VALDEF, ENDEF, RTDEF, COLON, FOR*). The first of these two words has the name's dictionary pointer (relative address of the name string in the dictionary) in the right-hand 12 bits, and the data type of the name in the left-hand 4 bits. The second word is used for various purposes, depending on the type.

2. Wherever a name is used as an expression, the tree node word which would normally be a pointer to the expression node is set to "#100000 + name"; that is, to the dictionary pointer for the name, with bit 0 + 1. [Since a tree node pointer always has bit 0 = 0, pointers to exp nodes can be distinguished from names by bit 0. When SAE encounters a name (bit 0 = 1) where it expects a pointer to an expression node, it looks up the name in the dynamic symbol table, and puts a pointer to the definition node in its place. See the "Dvec" documentation.]

LINE nodes

In most places where a "--stat" appears, the pointer will point at a *LINE* node; the third word of the *LINE* node points at the actual statement node. The *LINE* nodes allow the compiler to print the source text corresponding to a tree segment when an error is detected. The procedure WriteLine in *BCPL0* outputs a line of source text, given the (char number) from a *LINE* node. *LINE* nodes may also appear in *AND* nodes (*AND* nodes appear only under *LET* nodes.)

These are the two top-level routines in SAE; both in *BSAEI*.

Decllabels (address of a tree node)

Scans the tree branch looking for *COLON* nodes, calling DeclStatic to declare the static variables which are defined by such nodes. Decllabels just remembers some information, and uses Scanlabels to do the work. Scanlabels recurses on itself whenever it encounters a non-declaration node (a node which doe not start a new block) which points at a substatement branch of the tree.

Declvars (address of a tree node)

Scans the tree branch. It processes each declaration node, building the dynamic symbol table Dvec; an it calls Lookat whenever it encounters an expression. (Lookat replaces a name with a pointer to it definition.) Declvars calls Decllabels and then recurses on itself for nodes which have substatemen branches not examined by a higher-level instance of Declabels.

The only purpose of Decllabels is to implement the well-known Algol label kludge, which requires label defined within a block to be treated as it if were declared at the beginning of that bloc

So the real work of SAE is done by Declvars.

Dvec points at a vector allocated by the SAE driver DeclareNames in *BSAE0.* There are 2 words entry (DvecN = 2) and 512 entries max: (DvecMax = 512). There are three global indexes into Dv

Dvec!DvecS is the first unused entry.

Dvec!DvecE is the first entry made for the declaration node being processed. (That is, the first incomplete entry.) So searches begin at Dvec!(DvecE-2) and proceed backwards towards Dvec!0.

Dvec!DvecP is the first dynamic variable for the current function or routine. A referenced dynam variable (defined with let or as formal parameters) must be between Dvec!DvecP an Dvec!DvecS, since dynamic variables cannot be referenced outside of the procedure which they are declared.

These global indexes are saved whenever a declaration node is encountered. The declaration is processed; then the subtree under the declaration is processed; and finally the global indexes are restored. This is the mechanism whereby the block structure for the scope of names is maintained

There are two kinds of entries in Dvec: normal names (variables and manifests) and structure nar

Normal name: Dvec!i   # 100000 + name

Dvec!(i+1)   ---------------------   (type) + name

type-dependent extra word

"name" means the dictionary pointer assigned to the name string by LEX.

#100000 distinguishes normal names from structure names.

The second word is a pointer to the two free words allocated by the name. (In *STATIC, EXT,* and *MANIFEST* nodes, the words are in the node itself; in *VALDEF, FNDEF, RTDEF, COLON,* and *FOR* nodes, there is a pointer to the two-word block.) A dictionary pointer may be 12 bits wide; the high-order 4 bits indicate the data type of the name. The possibl types are:

LOCAL: dynamic variable (defined with "let name=val" or as a formal paramete

CONSTANT: manifest constant

LABEL
ZLABEL
EXTLABEL
ZEXTLABEL   (all state variable types.)
INTLABEL
ZINTLABEL

A name is declared as a static variable in an external declaration and/or in a static declaration, function, or routine declaration (let F(a,b)---), or statement label (L:---). In external and static declarations, a name may be preceded by "@", indicating that it is to be a page-zero static.

If a name appears in both an external declaration and one of the other kinds of sta declaration, its type is *INTLABEL,* or *ZINTLABEL* if it was declared to be in page zero.

If a name appears in an external declaration, but not in another declaration, its type is *EXTLABEL* or *ZEXTLABEL.*

If a name is declared as static, but not as external, its type is *LABEL* or *ZLABEL.*

The second word of the two-word name node contains:

LOCAL:	0, to be set to the frame offset by TRN
MANIFEST:	CAE leaves a pointer to the expression in this word; SAE evaluates it and replaces the expression pointer with the value.
other:	A static number is allocated ( ) and stored in this word. This number is used in the Ocode to indentify the variable.

Structure names in Dvec are intermixed with normal names. A structure name entry looks like:

Dvec!i	name
Dvec!(i+1)	ptr to structdef.

(Structure names can appear in the tree only in certain contexts (as the right-hand operand of *LEFTLUMP*, *RIGHTLUMP*, *SIZE*, and *OFFSET*). So normal name searches (via CellWithName) ignore nodes with bit 0 = 0, and structure name searches (with StrutWithName) ignore normal names.)

Only top-level structure names are entered in Dvec. The second word of a structure name entry is a pointer to the structure definition node in which the name is defined. This node can be a *FIELDLIST*, *OVERLAYLIST*, or *UPLUMP* node, or a two-word node which looks like:

#100000 + name

ptr to structdef/bitdef

(a "bitdef" node is a *BIT*, *BYTE*, or *WORD* node.)

The procedure DeclStruct in *BSAE2* creates structure entries in Dvec, and scans the structure definition for reasonability.

The procedure LookatQual in *BSAE4* processes structure references.

SAE must pass information to the code generator pass NCG about each static variable it defines. S
SAE opens the Ocode temp file *SSS.BC*, and outputs the following:

→For names of type: *LABEL, ZLABEL, INTLABEL, ZINTLABEL:*

        one byte: *LOC/ZLOC/INT/ZINT*
        two bytes: dictionary pointer
        one byte: "value type"
        two bytes: "value"

The "value type" word and "value" depend on how the name was defined.

a. If defined by static <name>=<exp>,

        value type = static
        value   = value of <exp> (it must be a ?-time constant)

b. If defined by let <name>=<formals> . . . (procedure)

        value type = *ENTRY*
        value   = a sequential number allocated for each *ENTRY* or *LENTRY.*

c. If defined by <name>:   (statement label)

        value type = *LENTRY*
        value   = a sequential number allocated for each *ENTRY* OR *LENTRY.*

→For names of type *EXTLABEL* or *ZEXTLABEL,*

        one byte: *EXT/ZEXT*
        two bytes: dictionary pointer

That is, external names which do not appear in a static, function, routine, or labe!
declaration have no value. The Ocode information for such names is output only if the
name is used in some expression.

When an SAE Ocode item is output for a name a sequential number is assgned to that name. Whe
TRN later outputs an Ocode item which refers to that name, it identifies the static name with th
number. *So the order in which static names are output by SAE is important.*

For procedure and label names, another sequential number is assigned by SAE. When TRN lat
generates an Ocode item for a procedure entry point or a label definition, it ouputs both the sequenti
static number (identifying the name) and the sequential entry number (identifying the entry point), :
that NCG knows what entry point to assign to each name.

The SAE Ocode is scanned by ScanImpures in *BNCG1.*

TRN scans the tree and outputs the remainder of the Ocode stream. The principal routines are Trans in *BTRN1*, which processes statement nodes; and Load in *BTRN5*, which processes expressions which Trans encounters in the tree.

Load (pointer to expression node):

Generates the appropriate Ocode to place the value of the expression on top of the simulated stack

If the expression pointer has bit 0 = 1, it is a pointer to a name declaration node. Load checks th type of the node (the left-hand 4 bits of the first word of the declaration node); according to type Load outputs a one-byte operator, followed by two 2-byte values:

LN value	dict name for a manifest name
LP frame-loc	dict name for a dynamic name
LL static number	dict name for a non-page-zero static name
LZ static number	dict name for a page-zero static name.

and increases SSP by one.

If the expression pointer has bit 0 = 0, it points at an expression node, so Load branches on the kind of node it is.

1. For binary operators; Load basically does

Load (left-hand expression)   [which increases SSP]

Load (right-hand expression) [which increases SSP]

output binary operator

decrease SSP

return

These operators are:

> *EQ NE LS LE GR GE*
>
> *MULT DIV REM LSHIFT RSHIFT*
>
> *PLUS MINUS LOGAND LOGOR EQV NEQV*
>
> *VECAP*

[For commutative binary operators and for relations, the right-hand operand is output first if it is a constant. (If both operands are constants, Lookat in SAE will have pruned the tree for most such nodes.)]

2. Other simple nodes:

NOT, NEG, RV	:	Load (operand) output *NOT, NEG,* or *RV* operator
TRUE, FALSE	:	output *TRUE, FALSE* operator
NUMBER	:	output *LN* operator output binary value (2 bytes)

[*LN* expects the name of a manifest after the value.]

CHARCONST	:	output *LC* operator
	:	output char code (2 bytes)

STRONGCONST	:	output *LSTR* operator
		output byte stream: length (1 byte)
		char 1 (1 byte)

char n (1 byte)

TABLE	:	output *TABLE* operator
		output table length (2 bytes)
		output table values: word 1 (2 bytes)

word n (2 bytes)

3. *LV*   Load calls LoadLv

The operand of an lv-expression must be a name, an rv expression, a vector subscript expression, a structure reference, or a conditional expression.

→For names, LoadLV outputs:

dynamic variable: *LLP* frame effect dict name

non-page-zero static: *LLL* static number dict name

page zero static: *LLZ* static number dict name

→For vector subscripts,

Load (left-hand operand)

Load (right-hand operand)

output *PLUS* operator

→For rv-expressions,

Load (operand)

output *LVRV* operator.

(This is done because "lv (rv x)" is not equivalent to "x" if bit 1 of "x" is set during execution. NCG generates a subroutine call to check this, and follow the indirect chain of "x".)

→For structure references, LoadLV processes the field qualifier, and then generates Ocode to divide the bit-offset of the field by 16, if necessary, and to add in any remaining constant word-offset.

→For "lv(<exp>?<exp1>,<exp2>)", LoadLV generates Ocode for "<exp>?lv<exp1>,lv<exp2>".

4. VALOF

Load calls Trans to process the statement operand of *VALOF*. A "results" statement in this block will cause Trans to output a Ocode operator which will in turn cause NCG to compile a jump to an internal label. After Trans has processed the statement, Load assigns this label (outputting *LAB*, internal label number) and then outputs *RSTACK* stackpointer, which indicates to NCG that a result value is to be found (in AC0) at this point in the program.

5. COND

Load generates Ocode for "<exp>?<exp1>,<ext←p2>" as if it were "valof [test<exp>then results<exp1> or results<exp2.]

For example, "x Is 0? -x, x" outputs the following:

```
LP x                          (load dynamic var "x")
LN 0                          (load constant "0")
LS              (less than operator)
JF labl                       (if false, jump to internal label)
LP x                          load "-x" and
NEG                            "    "    "
RES lab2                      (preserve result and jump to end of condition)
LAB labl                      (assign first label here)
LP x                          (load "+x")
RES lab2                      (preserve result and jump to end of condition)
LAB lab2                      (assign result label)
RSTACK p        (push a value onto the simulated stack at
                stack portion p)
```

6. FNAP

Load calls Transcall in *BTRN3*

Transcall also handles *RTAP* for Trans. *FNAP* and *RTAP* are different only in that *FNAP* appears in an expression context, hile *RTAP* appears in a statement context.

Ocode: *FNCALL/ RTCALL* (operator)

For each parameter:

```
Load(parameter i)
PARAM (operator)
parameter number (one byte)
total # of parameters (one byte)
dict name of procedure (two bytes,
    0 if not a name)

Load (procedure expr)
FNAP/ RTAP (operator)
total # of parameters (one byte)
stack position at beginning of call (two bytes)
```

7. Structure References

CAE

Rstruct(read overlay switch) in *BCAE4* parses structure declaration. It builds structdef nodes - see the tree node templates for the format of these nodes.

Rqualname(number of extra words in node) in *BCAE4* parses structure references. It builds a node containing the qualifier descriptor, with the specified number of words at the beginning (2 for *SIZE, OFFSET*; 3 for *LEFT/RIGHTLUMP*).

SAE

Declvars in *BSAE1* calls DeclStruct in *BSAE2* to scan structure declarations. It declares top-level names in Dvec, evaluates replication and size constants in the declaration, and replaces *RV* (@) names with pointers to the defining structure.

Lookat in *BSAE3* calls LookatQual in *BSAE4* to process *LEFTLUMP, RIGHTLUMP, SIZE,* and *OFFSET* nodes. LookatQual basically evaluates all field effects which do not involve non-constant subscripts, and constructs a representation of the field-access algorithm.

TRN

DoQual in *BTRN3* optimizes the algorithm given by LookatQual and generates the basic Ocode for the structure reference.

NCG

The routines in *BNCG8* handle most of the structure Ocode items. CGplus in *BNCG7* looks ahead for some structure Ocode items for optimization purposes.

BSAE4

LookatQual (addr of start of qualdesar in *LEFTLUMP, RIGHTLUMP, SIZE, OFFSET* ocode (see the template for these tree nodes)

This procedure creates a vector (in the tree space) describing the referenced field, and puts a pointer to it into the node (node!3 for *LUMP*'s, node!2 for *SIZE* and *OFFSET*).

This vector has the form:

```
0:      offset (in bits) after the last non-constant subscript
1:      size of field in bits
2:      number of non-constant subscripts
```

for each non-constant subscript:

```
offset (in bits) to this subscript
→exp for subscript
→(low limit value, high limit value)
size of replicated element
```

(Offsets for constant subscripts are computed and added in at compile time.)

So the algorithm for computing the bit number of the first bit of the accessed field of:

$$
\begin{array}{ll}
0: & K_0 \\
1: & S_0 \\
2: & n
\end{array}
$$

$$
\begin{array}{l}
K_i \\
\to X_i \qquad\qquad i = 1 \text{ to } n \text{ (n non-constant subscripts)} \\
\to (L_i, H_i) \\
S_i
\end{array}
$$

is $K_0 + \Sigma(K_i + (X_i - L_i)*S_i)$

(This is a run-time algorithm, since the X_i are not constants.)

DoQual in *BTRN3* optimizes this expression as it generates the Ocode for the structure reference.

BTRN3 DoQual(inputdescriptor, outputvector, wordsubscriptswitch)

inputdescriptor is the address of a vector generated by LookatQual in *BSAE4*.

$$
\begin{array}{lll}
\text{word } 0: & K_0 & \text{(final offset)} \\
1: & S_0 & \text{(size of referenced field)} \\
1: & n & \text{(no. of non-constant subscripts} \\
3\to 3+(4+n) \; : & K_i & \text{(offseti)} \\
& \to exp_i & \text{(ith subscript exp)} \\
& \to (L_i, H_i) & \text{(subscript limits)} \\
& S_i & \text{(size of replicated field)}
\end{array}
$$

outputvector

is a four word block for the final Ocode in formation. All preliminary Ocode needed for the computation of the non-constant subscript offsets is

generated. The outputvector contains:

> word 0: Ocode operator
> 1: constant part of offset computation in words
> 2: constant part of offset computation in bits
> 3: size of final field in bits.

wordsubscriptswitch

> if false, the subscript computation is carried out in terms of bits; if true, as much as possible is done in terms of words (bits/16). (The "bits" mode is used for *OFFSET* nodes.)

The possible Ocode operators in outputvector!0 are:

WQUAL : the field is a full word or a part of a single word
XQUAL : the field is $\leq$ 16 bits wide, but overlaps a word boundary.
MWQUAL : the field is n>1 words long, and starts on a word boundary.

(The only non-constant subscripts are on elements which are 16*n bits wide)

YQUAL : the field is 8 bits long and starts on a byte boundary
WBQUAL : the field is $\leq$16 bits wide, and a non-constant bit subscript is present
MWBQUAL : the field is >16 bits wide and a non-constant bit subscript is present

WQUAL:

> Load (wordaddress)
> wordoffset
> bitoffset (0-15)
> length (bitoffset + length <16)

XQUAL:

> Load (wordaddress)
> wordoffset = 0
> bitoffset (1-15)
> length (<16; but (bitoffset+length)$\geq$16)

YQUAL

> Load (bytepointer)
> wordoffset=0
> bitoffset=-
> length=i

MWQUAL

Load(wordaddress)
wordoffset
bitoffset=0
length=16*n   n>1

MWBQUAL

Load (wordaddress)
Load (bit offset)
wordoffset=0
bitoffset=0
length (≤16)

MWBQUAL

Load (wordaddress)
Load (bitaddress)
wordoffset=0
bitoffset=0
length (>16)

OCODE   (output by TRN)

S is the simulated stack. Ocode operators are one byte. Arguments to the following operators are two bytes, left followed by right, unless otherwise indicated.

LINE   charnum
PLINE   charnum

Inserted in the Ocode before most statements, so that NCG can print the corresponding source code on errors or on the ASM listing. *PLINE* appears only one for each line of source text; *LINE* appears for each statement on a source line.

LP   frame position   0/name

Push onto S the value of the dynamic variable allocated at this frame position.

LN   binary value   0/name

Push the binary value onto S.

LL & *LZ*   static var number   0/name

Push onto S the value of the static variable whose number is given. (*LZ*⟹ it is a page zero static).

LLP   frame position   0/name

Push onto S the address of the dynamic variable.

LLL & *LLZ*   static var number   0/name

Push onto S the address of the static variable. (*LLZ*⟹ page zero static)

LC   binary value

    Push the binary value onto S.

LSTR   ⟨byte stream⟩

    Push the address of a string constant onto S. The ⟨byte stream⟩ consists of one l giving the number of chars, followed by that number of bytes.

TABLE   ⟨double-byte stream⟩

    Push the address of a table constant onto S. The ⟨double-byte stream⟩ consists of bytes giving the number of binary values, followed by that number of 2-byte binai values.

TRUE

    Push the value #177777 onto S.

FALSE

    Push the value 0 onto S.

LLVP   frame offset

    Push onto S the address of the frame word which is ⟨frame offset⟩ words below th vector area of the frame. (See   ).

NEWLOCAL name

    Declare a new dynamic variable whose frame position is the current top of S. The value currently on the top of S is the value to be stored in the variable.

    "let x=10" generates   *LN* 10 0

"let v=vec 10" generates

NEWLOCAL (name of x)
LLVP <??>
NEWLOCAL (name of v)

SP frame offset 0/name

Store the value at the top of *S* into the dynamic variable whose frame offset is given. Pop *S*.

SL, SZ static var number 0/name

Store the value at the top of *S* into the static variable whose number is given. (*SZ*⇒page zero static) Pop *S*.

RV

Treat the value at the top of S as an indirect address word. Replace the top of S with the contents of the memory address at references.

VECAP

Treat the sum of the top two values on S as a memory address; replace those two values with the contents of that memory address.

STIND

Treat the value at the top of S as an indirect address word. Store into the location it references the value at (top-1) of S. Pop S twice.

STVECAP

Treat the sum of the top two values on S as a memory address; store into that location the value at (top-2) of S. Pop S three times.

Treat the value at the top of S as an indirect address word. Replace the top of S with the memory address that would be referenced by the indirect addr word. (I.e., follow the indirect address chain.)

"x=rv y"	"x=y!i"	"rv x=y"	"x!i=y"	x=lv(rvy)
LP y	*LP* y	*LP* y	*LP* y	*LP* y
RV	*LP* i	*LP* x	*LP* x	*LVRV*
SP x	*VECAP*	*STIND*	*LP* i	*SP* x

PLUS
MINUS Binary operators: replace the top two elements of S with the value
MULT   of the operator applied to their values. (The top element is the right-hand
DIV   operand.)
REM
LSHFT
RSHFT
LOGAND
LOGOR
EQV
NEQV

EQ
NE   Relations: replace the top two elements of S by the value #177777 if the
LS   relation is five, or by 0 if false.
LE
GR
GE

NEG   Binary operators: replace the top element with the result of the operator.
NOT

JUMP   internal label number

Transfer control the place where *LAB* <internal label number> appears in the Ocode.

JT   internal label number

Transfer control if the value at the top of S is non-zero. Pop the value off S.

JF   internal label number

Transfer control of the value at the top of S is zero. Pop the value off S.

RES   internal label number

Transfer control as for *JUMP*, after preserving the value at the top of S (i.e., load it into AC0). (Generated by "results<exp>") Pop the value off S.

LAB   internal label number

Assign the place to which the above should transfer control.

(*LAB's* are also output for "case" labels.)

GOTO

The value at the top of S is the address to which control should be transferred. Pop S.

FINISH

ABORT

End execution of the entire program. (On the Nova, do a .RTN system call.) (*ABORT* types a message on the terminal.)

number of cases	default label number
case value$_1$	case label number$_1$
.	.
.	.
.	.
case value$_n$	case label number$_n$

(2+n*2 parameters, each 2 bytes)

The value at the top of S is the value to be branched on. If this value is among the <case value$_1$> values, transfer control to the corresponding <case label number$_1$>. Otherwise, transfer control to the <default label number>. Pop the value off S.

(*LAB* ocode operators are output by TRN at each case label.)

LENTRY   static var number   entry point number

Generated when a statement label is encountered. Its two parameters are like the first two of *ENTRY*.

ENTRY   static var number   entrypoint number   MaxSSP   MaxVecSSP

The static variable is defined as a function or routine which begins here. The <entry point number> associates this entry point with the static variable Ocode information which was output by SAE.

MaxSSP is the size of the frame which will be needed by the code for this procedure, exclusive of the space for vectors. MaxVecSSP is the space needed for vectors. So the frame size is the sum of these.

SAVE   initial frame size

This Ocode generator appears only after an *ENTRY* operator. Its parameter is the number of formal parameters plus the number of words needed for the frame header.

NUMARGS   name

If a "numargs" variable was specified after the argument list, this Ocode operator causes it to be declared.

RTRN

Generated by "return", and at the end of a routine body. Causes a return from the current routine.

FNRN

Generated at the end of a function body. Causes a return from the function, with the value at the top of the stack S as the result value.

ENDFRAME

Marks the end of the procedure body begun by the corresponding *ENTRY*.

RTCALL
FNCALL

Signals the beginning of a procedure call. (*RTCALL* if standing alone as a statement, *FNCALL* if used in an expression.)

PARAM   i(one byte) n(one byte) 0/name(2 bytes)

The value at the top of S is the ith parameter to be passed to a procedure. A total of n arguments are to be passed. The name of the procedure to be called is given as the third parameter to *PARAM* (if the expression before the argument list is not a name, 0 is used).

RTAP

FNAP

        n(one byte) p(2 bytes)

The value at the top of S is the address of the first instruction of the routine or function to be called.

n arguments are to passed; the argument values are the values in (top-1) to (top-n) of S.

p is the offset of the top of S at the place in which the *RTCALL* or *FNCALL* appeared (it is included here for checking only).

If the operator was *FNAP*, the value of the function call is left on the top of S.

STACK   stack top position

Force the top of S to be at <stack top position>-1.

RSTACK   stack top position

Force the top of S to be at <stack top position>-1; then Push onto S the value that was preserved by the *RES* that was just executed.

        RES labnum is generated by "results". At the end of a "valof" block, the <labnum> is assigned (with LAB labnum). The next Ocode item after this *LAB* is *RSTACK*.

        So *RES* forces a value into AC0, generates a jump, and then forgets about the value it loaded. *RSTACK* pretends that some value has just been loaded into AC0.

The purpose of *STACK* (and *RSTACK*) is to indicate where the simulated stack top should be at the beginning and end of blocks; that is, it appears after declarations, and after the last statement of a block that began with a declaration. Basically, then, *STACK* (and *RSTACK*) are used to keep TRN and NCG synchronized with respect to what the value of "SSP" should be at critical points. Certain other Ocode items are also accompanied by an "SSP" value for the same reason.

STORE

This Ocode operator is output in order to force NCG to generate code to store into actual frame locations any values on the simulated stack which have not been fully processed.

        STORE should not really be an Ocode item; NCG should know when it is necessary to clean up the simulated stack. The operator is left over from TX-2 days.

argvec is used as a stack with five words per entry. Each entry describes an item on the simulated stack; that is, a run-time operand. The NCG variables arg1, arg2, and arg3 always point at the top entry, the entry next to the top, and the entry below that, respectively; when an entry is pushed or popped, these pointers are changed. So arg1, arg2, and arg3 are pointers to the three most recent run-time operand descriptors which have been processed.

An argvec entry consists of five words; if arg is a pointer to some argvec entry, its fields are referenced with type!arg, loc!arg, ref!arg, pos!arg, and name!arg.

type!arg indicates what kind of operand is being described;
loc!arg depends on the type. The possible types, and the meaning of the corresponding loc's are:

LOCAL   frame offset

> Word <frame offset> in the frame of the current procedure. This describes both dynamic variables and temporary cells used in the frame.

LABEL   static variable number

> Static variable. The <static variable number> is the number used throughout SAE, TRN, and NCG to identify the static.

COMMON static variable number

> Page zero static variable. The <static variable number> is as for *LABEL.*

NUMBER   binary value

> Constant with the given value.

RVLOCAL   frame offset

> Describes an operand which is to be referenced by indirection through the <frame offset>th word of the frame. This descriptor would be generated, for instance, by an operand "rv x", where "x" is a dynamic variable.

RVLABEL   static variable number

> Describes an operand which is to be referenced by indirection through the static variable corresponding to the <static variable number>.

RVCOMMON static variable number

> Like *RVLABEL,* but the static is a page-zero static variable.

RVNUMBER binary value

> Results from "rv n", where "n" is a constant expression. This operand is to be referenced by indirect through a memory word which contains the <binary value>.

LVLABEL   static variable number

> Describes an operand whose value is the address of the static variable.   Results from "lv s", where "s" is static.

LVCOMMON static variable number

> Like *LVLABEL*, but the static is a page-zero static variable.

AC   0

> Describes an operand which is in AC0 or AC1.

XR   3

> Describes an operand which is in AC3.

Temporarily ignoring the other fields of an arg, let us look at an example of how NCG processes a statement.

Assume that p (in word 6 of the frame) is a dynamic variable and s (static var number 13) is a static.   Then the statement

$$s = rv \ p + 2$$

would generate this Ocode:

> *LP*   6
> *RV*
> *LN*   2
> *PLUS*
> *SL*   13

NCG proceeds as follows:

LP 6

> causes the descriptor *LOCAL*, 6 to be pushed onto argvec. So arg1 points at this descriptor.

RV

> is a unary operator, so it is to be applied to the operand described by *arg1*. This changes the descriptor in *arg1* to RVLOCAL, 6.

LN 2

> causes the descriptor NUMBER, 2 to be pushed onto argvec. So now:

> > arg2 → *RVLOCAL*, 6
> > arg1 → *NUMBER*, 2

PLUS

> is a binary operator, so it is to be applied to arg2 and arg1. This will cause the code generator to generate the necessary code to load the operands described by arg2 and arg1 into free accumulators, and then generate an ADD instruction:

> > LDA 0 @6,2   //rv of frame word 6.
> > LDA 3 .+K1   //.+K1 will contain a 2.
> > ADD 3,0   //add, leave, result in AC0.

(In the process of generating this code, arg1 and arg2 are modified several times; we will ignore this in this example, since they will go away immediately.)

After this code is generated, argvec will be popped twice, and then the descriptor AC, 0 will be pushed. So now

arg2 → ?
arg1 → AC, 0.

SL 13

indicates that the operand described by arg1 is to be stored into static variable #13. So NCG will, upon encountering this ocode item, push a descriptor *LABEL* 13, giving:

arg2 → AC, 0
arg1 → LABEL, 13

and then generate code to store arg2 into the operand described by arg1. (If arg2 were not in an accumulator code would just have to be generated to lead it.) So

STA 0 @.+K2 ·   //.+K2 will contain
                       //the address of s

Now argvec is poped twice, and we are done.

Note that no code was generated until the *PLUS* was seen, because the operands up to that point could be described by argvec descriptors. One of the purposes of the argument stack is to allow the generation of code for an operand to be postponed as long as possible.

The name field of an argvec entry contains the dictionary pointer for the name of the operand, or 0 if it is not a variable or manifest. The name field is used only to generate meaningful comments on the /A listing, if it is requested.

The ref word in argvec entries allows more complex operands to be described in argvec, so that code generation can be postponed even further. If arg is a pointer to an argvec entry, ref!arg is interpreted as follows:

if ref!arg is 0, the operand is as described above.

if bits 0+1 of ref!arg are 01, the operand resulted from a vector subscript expression; where one operand of "!" was a constant between -#200 and +#177. The right-hand 8 bits of ref!arg contain the constant subscript. For example,

LOCAL, loc = 6, ref = 0 describes an operand
  which is in word 6 of the current frame; it is loaded into AC0 with
  LDA 0, 6, 2.

LOCAL, loc = 6, ref = #40007 describes an operand which is the result of an expression like "v!]", where "v" is the dynamic variable which is allocated to word 6 of the frame. This operand is loaded into AC0 with:
  LDA 3 6,2
  LDA 0 7,3

if bits 0+1 of ref!arg are 11, the operand resulted from "rv" applied to a vector subscript expression where one operand of "!" was a small constant.

LOCAL, loc = 6, ref = #140007 results from "rv(v!7)", and is loaded into AC0 by
  LDA 3 6,2
  LDA 0 @7,3

The only legitimate values for a ref field are:

0   normal addressing

#40000+n   subscripted reference:

> n is an 8-bit signed value,
> so the ref field for "v!-7"
> would be #40371.

#140000+n rv of a subscripted reference.

Note that #100000+n is not legitimate; bit 0, indicating indirection, is only set if the operand is already subscripted. The only reason for having bit 1 set is that n may be 0. E.g., the ref field for "v!0" is #40000.

A descriptor of type *AC* may not have a non-zero ref field. If a *VECAP* Ocode operator is encountered with

$$arg2 \rightarrow \textit{AC}, \; loc = 0, \; ref = 0$$
$$arg1 \rightarrow \textit{NUMBER}, \; loc = 7, \; ref = 0.$$

(e.g., resulting "(v+w*3)!7" after (v+w*3) is loaded into AC0). NCG will generate

MOV 0, 3

then pop the two args and push XR, loc = 3, ref = #40007. If this operand is to be loaded into AC 0, it will be done with

LDA 0 7,3

The pos field of an argvec entry records the offset in the current frame of the word which is reserved as a temporary call for the operand described by the entry. An example:

The Ocode for: x- (a+b)/(c*d) is

LP x
LP a
LP b
PLUS   x,a,b,c,d mean the
LP c   frame offsets for these
LP d   variables.
MULT
DIV
MINUS

By the time *MULT* is encountered, argvec contains

```
----  →  LOCAL x,
arg3  →  AC 0
arg2  →  LOCAL c
arg1  →  LOCAL d
```

and the code

```
LDA 0 a,2
LDA 3 b,2
ADD 3 3,0
```

has been generated.

Now code must be generated for multiplying c and d; this will require using both AC's, so a temp is needed for arg3. Cells in the frame are used for such temps; pos!arg3 contains the offset of the temp for arg3.

Assume that pos!arg3 = 11 (See below for how pos gets set)
So the *MULT* will generate:

```
STA  0  <11,2
LDA  0  c,2
LDA  1  d,2
JSR  multiply routine  //leaves result in AC1
```

When the store into temp gets generated, the descriptor for arg3 is changed to *LOCAL*, loc = 11. So after the multiply popped arg2 and arg1, and pushes the result descriptor *AC*, 1, argvec contains

```
arg3  ->  LOCAL x
arg2  ->  LOCAL 11
arg1  ->  AC 1
```

Now *DIV* must load arg2 into AC0 before doing the division; since arg2 is *LOCAL*, this is done by

```
LDA  0  11,2
```

Then the rest of the code is generated by *DIV* and *MINUS*:
```
JSR  divide subroutine   //leaves result in AC0

LDA  3  x,2
SUB  3,0
```

and the final argvec is

```
arg3  ->  ?
arg2  ->  ?
arg1  ->  AC 0.
```

Any operand in argvec may need to be stored in a temp at any time, so a frame word is reserved for each operand which is stacked up in argvec. Frame space is, of course, also used for dynamic variables; this variable space is allocated and de-allocated according to the lexical scope of declarations.

```
let f(a,b,c,d) be
[let x = - - -
      [let p,q = - - -

      ]
      let y = - - -
      x = x- (a+b)/(c*d)      //same expression as in the
                              //example above.

]
```

The frame for this procedure looks like

```
0
1          frameheader
2
3
4          a
5          b
6          c
7          d
8          x
9          p   y
10         q
11         -      temps which might be needed
12         -      for processing x-(a+b)/(c*d)
13         -
```

. In the inner block, words 9+10 are reserved for p and q. When this block is exited, they are unreserved. Then word 9 is reserved for y. So at the beginning of the final assignment statement, the first unused word of the frame is word 10; this is where temps needed for this expression will be allocated. When the *MULT* ocode operator is encountered, argvec will contain:

	type	loc	ref	pos	
---- →	*LOCAL*	8	0	10	//x
arg3→	*AC*	0	0	11	//(a+b)
arg2→	*LOCAL*	6	0	12	//c
arg1→	*LOCAL*	7	0	13	//d.

So word 11 of the frame will be used as the temp for saving arg3.

An aside on frame size determination:

The frame size needed for this procedure would be (at least) 14 words, since word 13 of the frame was reserved as a temp for "d" in computing "x-(a+b)/(c*d)". This is the case even if, as in the above, that word is never actually needed. (The reason is that by the time NCG finds out that it is not needed, it is too late to do anything about it; TRN determines how the frame space is to be allocated.)

If a procedure allocates vector space ("let v = vec 100") the variable ("v") is allocated among the other dynamic variables and temps; but the vector space (the 101 words that "v" will point to) is allocated below the last temp word reserved. So if the declaration of "y" in the above example were "let y = vec 100", "y" would still be word 9, and would point to word 14 of the frame at run-time. The frame size would be 115 words (14+101).

SSP is the global NCG variable which keeps track of allocation of words in the procedure frame. The value of SSP is the offset of the first unused frame word relative to word 0 of the frame. Thus SSP always has the value pos!arg1+1.

SSP is initialized by case *ENTRY*. of the procedure Scanpures in *BNCG2*. (The *ENTRY* ocode item is followed by a *SAVE*, whose 2-byte parameter is the initial value of SSP). Initstack [in *BNCG3*] is called, which sets up argvec and initializes SSP to 4 + (number of formal parameters to the procedure). At this point, argvec has three dummy entries, corresponding to arg3, arg2, and arg1; each is *LOCAL*, with loc, and pos fields set to SSP-3, SSP-2, SSP-1 respectively.

Push (type, ref, loc) pushes an entry onto argvec and bumps SSP. Pop1() and Pop2() pops one or two entries and decrements SSP.

SetSSP(newvalue) sets SSP to a new value. If the new value is smaller than the current value, it does the appropriate number of pops. If the new value is greater than the current value, it pushes the appropriate number of temp cell descriptors.

Clearstack(SSPvalue) generates code to store into temps every operand below the SSPvalue.

This routine generates the memory reference instruction op for the operand described by arg. It compiles any preliminary code needed to set up the reference (e.g., loading AC3 with the base address for a subscript reference), computes the address (indirect, index, and displacement fields) for the instruction, and finally generates (op + address).

op may be:

JMP, JSR, ISZ, DSZ
LDA0, LDA1, LDA2, LDA3
STA 0, STA 1, STA 2, STA 3
(that is, for LDA+STA, the AC field is part of op)

arg is a pointer to a five-word operand descriptor, usually in argvec (i.e., arg1, arg2, arg3). The descriptor may not be of type AC; it may be of type XR only if reflarg is non-zero. (That is, arg must describe a memory reference).

CG memref compiles the following code:

	Type!arg	loc!arg	ref!arg	code
x	LOCAL	p	0	op p,2
x!n	LOCAL	p	#40000+n	LDA 3 p,2; op n,3
@(x!n)	LOCAL	p	#140000+n	LDA 3 p,2; op @n,3
@x	RVLOCAL	p	0	op @p, 2
(@x)!n	RVLOCAL	p	#40000+n	LDA 3@p,2; op n,3
@((@x)!n)				
	RVLOCAL	p	#140000+n	LDA 3@p,2; op @n,3 [p is the offset of the word in the frame.]

(s is a static)

s	LABEL	l	0	op @.+N
s!n	LABEL	l	#40000+n	LDA 3@.+N; op n,3
@(s!n)	LABEL	l	#140000+n	LDA 3@.+N; op @ n,3 [.+N contains the address of the static]
@s	RVLABEL	l	0	op @.+N
(@s)!n	RVLABEL	l	#40000+n	LDA 3 @.+N; op n,3
@((@s)!n)				
	RVLABEL	l	#140000+n	LDA 3 @.+N; op@n, 3 [.+N contains the address of the static + #100000]

Type!arg	loc!arg		ref!arg	code

(z is a page-zero static)

z	COMMON	1	0	op a
z!n	COMMON	1	#40000+n	LDA 3 a ; op n,3
@(z!n)	COMMON	1	#140000+n	LDA 3 a ; op @n,3
@z	RVCOMMON	1	0	op @ a
(@z)!n	RVCOMMON	1	#40000+n	6DA 3 @a ;op n,3
@((@z)!n)	RVCOMMON	1	#140000+n	LDA 3 @a ;op @n,3

[a is the address
of the static in
page zero.]

(k is a number, or a manifest name)

k	NUMBER	k	0	op .+N
k!n	NUMBER	k	#40000+n	LDA 3 .+N; op n,3
@(k!n)	NUMBER	k	#40000+n	LDA 3 .+N; op@n,3

[.+N contains
the value k.]

---

lv s	LVLABEL	1	0	op .+N
	LVCOMMON	1	0	
(lv s)!n	LVLABEL	1	0	LDA 3 .+N ; op n,3
	LVCOMMON	1	#40000+n	
@((lv s)!n)	LVLABEL	1	#140000+n	LDA 3 .+N ; op @n,3

[.+N contains the
address of the static
or page zero static.}

CGmemref handles *RVNUMBER* specially if ref!arg is 0.

1. If the value of the constant is between 0 and #377:

@k	RVNUMBER	k	0	op k [i.e., like a page zero reference]

2. If the value of the constant is between #100000 and #100177, and not if compiling for Alto.

@k	RVNUMBER	k	0	op @ (k S#377)

[i.e., in indirect
reference through
a page zero location.

Otherwise,

	RVNUMBER	k	0	op @ .+N
	RVNUMBER	k	#40000+n	LDA 3 @.+N; op n,3
	RVNUMBER	k	#140000+n	LDA 3 @.+N; op@n,3

[.+N contains the
value k]

Note:

The operand type *RVLABEL* is never generated if compiling for the Alto (because "op @.+N", where ".+N" contains #1----- + address of static does not do a multiple indirection, as it does on the Nova). Instead, "@ s", where s is a static, generates

$$\text{LDA 3, @.+N} \quad \text{//.+N contains addr of s.}$$
$$\text{op 0,3}$$

The more complex cases ("(@s)!n" and "@@s!n)") are done by

$$\text{LDA 3 @.+N}$$
$$\text{STA 3 temp.2}$$

and setting the arg to *RVLOCAL*, temp. <ref>

The principal routines which manipulate the address modes of operands on argvec are CGrv() and CGsubscr(j), both in *BNCG4*.

CGrv() applies the "rv" operator to arg1.

CGsubscr(j) essentially does (arg2!(arg1+j)), where j is a number between #-200 and #177.

The PLUS Ocode operator also manipulates addressing modes (CGplus in *BNCG7*). It looks ahead at the next operator; if a *VECAP*, *STVECAP*, *WQUAL*, or *STWQUAL* operator is next, it tries to do things in an order which will generate reasonable code. E.G., CGplus does "(v+1)!i" as if it had been written "(v+i)!1".

Also, the structure code generation routines in *BNCG8* do a considerable amount of address-mode manipulcation.

CGmemref (op,arg) [in *BNCG4*]

This routine generates the memory reference instruction op for the operand described by arg. It compiles any preliminary code needed to set up the reference (e.g., loading AC3 with the base address for a subscript reference) computes the address (indirect, index, and displacement fields) for the instruction, and finally generates (op + address).

op may be:

JMP, JSR, ISZ, DSZ, LDS 0, LDS 1, LDA 3, STA 0, STA 1, STA 2, STA 3

(That is, for LDA + STA, the AC field is part of op.)

arg is:

a pointer to a five-word operand descriptor, usually in argvec (i.e., arg1, arg2, arg3). The descriptor may not be of type *AC*; it may be of type *XR* only if ref!arg is non-zero. (That is, arg must describe a memory reference).

CGmemref compiles the following code:

	type!arg		loc!arg	ref!arg	code
x	LOCAL	p		0	op p,2
x!n	LOCAL	p		#40000+n	LDA 3 p,2 ; opn,3
@(x!n)	LOCAL	p		#40000+n	LDA 3 p,2 : op@n,3
@x	RVLOCAL	p		#0	op @p,2
(@x)!n	RVLOCAL	p		#40000+n	LDA 3 @p,2;op@n,3
@((@x)!)	RVLOCAL	p		#40000+n	LDA 3 @p,2;op@n,3 [p is the offset of the word in the frame.]

(s is a static)

	type!arg		loc!arg	ref!arg	code
s	LABEL	1		0	op @.+N
s!n	LABEL	1		#40000+n	LDA 3 @.+N; op n,3
@(s!n)	LABEL	1		$40000+n	LDA 3 @.+N; op @n,3 [.+N contains the address of the above]
@s	RVLABEL	1		0	op @.+N
(@s)!n	RVLABEL	1		#40000+n	LDA 3 @.+N; op n,3
@((As)!n)	RVLABEL	1		#40000+n	LDA 3 @.+N; op@n.3 [.+N contains the address of the static + #100000]

(z is a pure zero static)

	type!arg		loc!arg	ref!arg	code
z	COMMON	1		0	op a
z!n	COMMON	1		#40000+n	LDA 3 a ; op n,3
@(z!n)	COMMON	1		#40000+n	LDA 3 a ; op @n,3
@z	RVCOMMON	1		0	op@a
(@z)!N	RVCOMMON	1		#40000+n	LDA 3 @a; op n,3
@((@x)!n	RVCOMMON	1		#40000+n	LDA 3 @a ; op@n,3 [a is the address of the static in page zero]

(k is a number, or a something name)

	type!arg		loc!arg	ref!arg	code
k	NUMBER	k		0	op .+N
k!n	NUMBER	k		#40000+n	LDA 3 .+N; op n,3
@(k!n)	NUMBER	k		#40000+n	LDA 3 .+n ; op @n,3 [.+N contains the value k]
lv s	LVLABEL	1		0	op .+N
	LVCOMMON	1		0	
(lv s)!n	LVLABEL	1		#40000+n	LDA 3 .+N ; op n,3
	LVCOMMON	1		#40000+n	
@(lv s)!n)	LVLABEL	1		#140000+n	LDA 3.+N ; op@n,3
	LVCOMMON	1		#140000+n	[.+N contains the address of the static or page-zero static.]

CGmemref handles *RVNUMBER* specially if ref!arg is 0.

(1)   if the value of th constant is between 0 and #377:

@ k   *RVNUMBER*   k   0   op k
                                                        [i.e., like a page-
                                                        zero static]

(2) if the value of the constant is between #100000 and #100377, and if not compiling
for Alto.

@ k   *RVNUMBER*   k   0   op @ (k S #377)

             [i.e., in indirect reference through a page-zero location.]

Otherwise,

             RNNUMBER   k   0   op @ .+N
             RVNUMBER   k   #40000+n   LDA 3 @.+N ; op n,3
             RVNUMBER   k   #140000+n LDA d @.+N ; op @n,3
                                                       [.+N contains the
                                                       value k]

Note: The operand type *RVLABEL* is never generated if compiling for the Alto (because
"op @.+N", where ".+N" contains "#100000 + address of static" does not do a multiple
indirection, as it does   on the Nova).   Instead, "@s", where s is a static, generates

        LDA 3, @.+N //.+N contains addr of s.
        op 0,3

The more complex cases ("@s)!n" and "@@s!n)") are done by

        LDA 3 @.+N
        STA 3 temp, 2

and setting the arg to *RVLOCAL*, temp,<ref>.

The principal routines which manipulate the addressing modes of operands on argvec are
CGrv() and CGsubscr(j), both in *BNCG4*.

        CGrv() applies the "rv" operator to arg1.

        CGsubscr(j) essentially does (arg2!(arg1+j)), where j is a number between #-200 and
        #177.

The *PLUS* Ocode operator also manipulates addressing modes (CGplus in *BNCG7*). It looks
ahead at the next operator; if a *VECAP*, *STVECAP*, *WQUAL*, or *STWQUAL* operator is next,
it tries to do things in an order which will generate reasonable code. E.g., CGplus does
"(v+1)!i" as if it had been written "(v+i)!1".

Also, the structure code generation routines in *BNCGS* do a considerable amount of address-mode manipulation.