1033

Bell Laboratories **Cover Sheet for Technical Memorandum**

Title- **Programming in LIL: A Tutorial**

Date- **June 17, 1974**

TM- **74-1352-6**

Other Keywords- **PDP-11**
Implementation Languages

Author	Location	Extension	Charging Case- 39394
P.J. Plauger	**MH 7C-205**	**X3644**	Filing Case- 39394

ABSTRACT

LIL is a Little Implementation Language for PDP-11 computers, suitable for writing system level code or in any situation where assembly-language coding is traditionally called for. A LIL compiler is available for use under the UNIX operating system. The object code produced is compatible with, and may be freely intermixed with, that produced by the UNIX assembler, Fortran, or C compiler.

This document provides a tutorial introduction to programming in LIL. A knowledge of machine level coding on the PDP-11 is assumed, and some knowledge of UNIX operating procedures is required to use the compiler. More complete information on LIL is provided in the reference manual, TM 74-1352-8.

Pages Text 18	Other 0	Total 18	
No. Figures 0	No. Tables 0	No. Refs. 0	

SEE REVERSE SIDE FOR DISTRIBUTION LIST

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM F-1328)

10 REFERENCE COPIES

+BOYD,GARY D
+BUCHSBAUM,S J
CHRISTENSEN,C
+CLOGSTON,A M
+CONDON,J H
CUTLER,C CHAPIN
FREENY,S L
GILLETTE,DEAN
GIORDANO,PHILIP P
+HANNAY,N B
+JENSEN,PAUL D
KEEFAUVER,W L
+KLEIN,MISS R L
+KOLETTIS,N J
LIMB,J O
LYCKLAMA,HEINZ
MCDONALD,H S
MILLER,S E
+NINKE,WILLIAM H
NOOLANDI,J
+OMOHUNDRO,WAYNE E
+PATEL,C K N
PRIM,ROBERT C
+REID,ROBERT E JR
ROBERTS,CHARLES S
+SLICHTER,W P
TEWKSBURY,S K
THOMPSON,JOHN S
TILLOTSON,L C
+WOLONTIS,V MICHAEL
YAMIN,MRS E E
+YOUNG,JAMES A
32 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

8 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

ABRAHAM,STUART A
ACKERMAN,A F
AHO,A V
AHRENS,RAINER B
ALCALAY,DAVID
ALLES,HAROLD G
AMRON,I

COVER SHEET ONLY TO

ANDERSON,M M
ARMBRUSTER,MISS M E
ARNOLD,GEORGE W
ARNOLD,S L
ATAL,B S
BADURA,DENNIS C
BALDWIN,GARY L
BARTLETT,WADE S
BASEIL,RICHARD J
BAUER,MISS H A
BAUGH,C R
BERGLAND,G DAVID
BERING,D E
BERNSTEIN,LAWRENCE
BEYER,JEAN-DAVID
BILOWOS,RICHARD M
BIREN,MRS IRMA B
BISHOP,MISS V L
BLINN,JAMES C
BLUE,J L
BLY,JOSEPH A
BODEN,F J
BOHACHEVSKY,I O
BOWYER,L RAY
BOYCE,W M
BRANDT,RICHARD B
BREWER,F A
BROWN,W STANLEY
BROWN,WILLIAM R
BULLEY,RAYMOND M
BURR,STEFAN A
BUTZIEN,PAUL E
CAMPBELL,STEPHEN T
CARAWAY,R E
CASPERS,MRS BARBARA E
CAVINESS,JOHN D
CEMASHKO,FRED
CHAMBERS,J M
CHAMBERS,MRS B C
CHEN,EDWARD
CHEN,STEPHEN
CHERRY,MS L L
CHODROW,MARK M
CLAYTON,DANIEL P
CLIFFORD,ROBERT M
COBEN,ROBERT M
COHEN,HARVEY
COLE,LOUIS M
COLLIER,ROBERT J
COOPER,A E
CORASICK,MISS M J
COULTER,J REGINALD
COURTNEY PRATT,J S
CRUME,LARRY L
D ANDREA,MRS LOUISE A
DAVIS,R L JR
DEUTSCH,DAVID N
DICKMAN,B N
DIMINO,L A
DIMMICK,JAMES O
DOLOTTA,T A
DRAKE,MRS L
DRYDEN,JOHN J

COVER SHEET ONLY TO

EDMUNDS,T W
EIGEN,D
ELLIOTT,R J
ELY,T C
EPDLE,K W
ESSERMAN,ALAN R
FABISCH,MICHAEL P
FARGO,GEORGE A
FELDMAN,STUART I
FELS,ALLEN M
FIGLIUZZI,MISS M E
FISCHER,H B
FLANAGAN,J L
FLEISCHER,HERBERT I
FONG,KENNETH T
FORT,JAMES,W
FOX,PHYLLIS
FOX,R T
FOY,J C
FRANK,MISS A J
FRANK,RUDOLPH J
FRASER,A G
FREEMAN,K GLENN
FRETWELL,LYMAN J
FROST,H BONNELL
FULTON,ALAN W
GARCIA,R F
GATES,G W
GAY,FRANCIS A
GEARY,M J
GELBER,MISS CHERON L
GEPNER,JAMES R
GEYLING,F T
GIBB,KENNETH R
GILBERT,MRS HINDA S
GIMPEL,JAMES F
GITHENS,JOHN A
GLASSER,A
GLUCK,F
GOGUEN,MS NANCY
GOLABEK,MISS R
GOLDSTEIN,A JAY
GORMAN,JAMES E
GRAHAM,R L
GREENSPAN,S J
GROSS,ARTHUR G
GUERRIERO,JOSEPH P
HAFER,E H
HALL,ANDREW D JR
HALL,MILTON S JR
HALL,W G
HAMILTON,PATRICIA
HAMMING,R W
HANSEN,R J
HARASYMIW,J
HARKNESS,MRS CAROL J
HARRINGTON,T DORSEY
HARRISON,NEAL T
HARUTA,K
HAUSE,A D
HEATH,SIDNEY F III
HEMMETER,RICHARD W
HEMMING,C JR

COVER SHEET ONLY TO

HERGENHAN,C B
HEROLD,JOHN W
HESS,MILTON S
HINES,MISS P E
HOLTMAN,JAMES P
HONIG,W L
HOYT,WILLIAM F
HUNNICUTT,CHARLES F
IPPLAND,FREDERICK C
IPPOLITT,O D
IRVINE,M M
IVIE,EVAN L
JACKOWSKI,D J
JACOBS,H S
JARIELSKI,C E
JAMES,DENNIS R
JESSOP,WARREN H
JOHNSON,STEPHEN C
JOHNSTON,W DEXTER JR
JOYNT,L
JUDICE,CHARLES W
KAPLAN,M M
KAPLAN,ROBERT
KAYEL,R G
KEARNEY,ROBERT
KELLY,L J
KENNEDY,ROBERT A
KERNIGHAN,BRIAN W
KERTZ,DENIS P
KILLMER,JOHN C JR
KNOWLTON,KENNETH
KNUDSEN,DONALD B
KORNEGAY,R L
KREIDER,DANIEL W
KRUSKAL,JOSEPH B
LEE,WILLIAM C Y
LEGENHAUSEN,S
LESK,MICHAEL E
LESK,MS ANN B
LIEBERT,THOMAS A
LINDERMAN,T
LINNEWEH,LOUIS H
LIN,SHEN
LUMMIS,ROBERT C
LUTZ,KENNETH J
MADDEN,MRS D M
MAHLER,G R
MALCOLM,J A
MALLOWS,COLIN L
MARNSELL,HENRY I G
MC CABE,PETER S
MC CRACKEN,MARYLOU
MC CUNE,R F
MC EOWEN,JAMES R
MC GILL,ROBERT
MC GONEGAL,MISS C A
MC ILROY,M DOUGLAS
MC MULLEN,E C
MC RAE,JEAN E
MC TIGUE,G E
MENIST,DAVID B
MENON,P R
METZ,ROBERT F

COVER SHEET ONLY TO

MILLER,ALAN R
MILLS,MISS ARLINE D
MOLINELLI,JOHN J
MOLTA,J W
MORGAN,DENNIS J
MORGAN,S P
MORRIS,ROBERT
MORR,PHILLIP L
MORATORI,RICHARD D
NEARLICH,W R
NELSON,N-P
NORTON,HERBERT O
NOWITZ,D A
O CONNELL,T F
O SHEA,W T
O SULLIVAN,JOHN A
OLSEN,RONALD G
OPPERMAN,D C
OSSANNA,J F JR
OTTO,JOHN B
OWENS,MRS G L
PALM,RICHARD C JR
PEARLMAN,R
PENNINO,T
PERITSKY,MARTIN M
PETERSON,RALPH W
PETERSON,T G
PFISTER,ROBERT G
PILLA,MICHAEL A
PINSON,ELLIOT N
PIRZ,FRANK
PITTS,CHARLES J
POLLAK,HENRY O
POPPER,C
RAACK,GERALD A
REED,S C
REHERT,ALLEN F
RIDDLERERGER,C O
RITACCO,J F
RITCHIE,DENNIS M
ROCHKIND,M T
RODIHAN,MRS PATRICIA A
RODRIGUEZ,ERNESTO T
ROSENTHAL,CHARLES W
ROSLER,LAWRENCE
ROVENO,MRS HELEN D
RYDER,J
SATZ,T R
SCHRYER,N L
SCHUPTER,W H
SCHWENKER,JOHN P
SPARS,EDWARD P
SPRINO,W T
STEMMELMAN,C T.
STRANK,J C
SHANK,R A
SHIPLEY,EDWARD N
SHORTER,J W
SIMON,DAVID J
STROWITZ,NORMAN B
SLAUGH,J H
SMITH,D W
SNARR,R C

* NAMED BY AUTHOR > CITED AS REFERENCE SOURCE

316 TOTAL

MERCURY DISTRIBUTION.....

COMPLETE MEMO TO:
10-EXD 13-DIR 135-DPH

COVER SHEET TO:
127
COPLAS = COMPUTING/PROGRAMMING LANGUAGES/ASSEMBLY

PADY,J E: MR 78201:

TM-74-1352-6 TOTAL PAGES 30

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.

PLEASE SEND A COMPLETE COPY TO THE ADDRESS SHOWN ON THE OTHER SIDE
NO ENVELOPE WILL BE NEEDED IF YOU SIMPLY STAPLE THIS COVER SHEET TO THE COMPLETE COPY.
IF COPIES ARE NO LONGER AVAILABLE PLEASE FORWARD THIS REQUEST TO THE CORRESPONDENCE FILES.

Bell Laboratories

Subject- **Programming in LIL: A Tutorial**

Date- **June 17, 1974**

From- **P. J. Plauger**

TM- **74-1352-6**

MEMORANDUM FOR FILE

LIL (a Little Implementation Language) is designed to help you write machine level code for the PDP-11. It looks like a high level language, because it is one (very much like C in fact); but it deals directly with registers, indexing, and all the other things you must keep in mind while writing assembly code. The major advantages it offers over assembly language are that you are encouraged to do some things in ways that have proved to be more reliable, and that the final program is very readable.

1. Expressions

We begin with simple expressions, which form the meat of a program. The PDP-11 code to form the sum of x, y, and z in register 0, and the sum of x and y in w is:

```
mov x,r0
add y,r0
mov r0,w
add z,r0
```

In LIL you could write

```
r0 = x;
r0 + y;
w = r0;
r0 + z;
```

but you are also allowed to say

```
r0 = x + y -> w + z;
```

or even

```
w = (r0 = x + y); r0 + z;
```

to get the same effect.

The idea is that expressions are evaluated left to right. Everytime you see the sequence

operand operator operand

the appropriate PDP-11 instruction is generated and **operator operand** is discarded, leaving just the left operand for further use. All operators are of equal binding strength in LIL — assignments (moves), adds and multiplies are produced in the order you specify. Only the use of parentheses will modify this strict left-to-right evaluation, as in the last example. There the rule is — evaluate any expressions inside parentheses, first for the left operand, then for the right, and take as

the operand the leftmost thing inside the parentheses. This rule can be applied to any depth of nesting.

There are a number of different operators defined in LIL, enough to let you specify any data manipulation instruction in the PDP-11. All of the operators are tabulated toward the end of this memo. Meanwhile, we will just explain any new ones as we happen across them.

2. Conditions

To get a little fancier, let us compute the maximum of x and y in r0:

```
        mov x,r0
        cmp r0,y
        bge 1f
        mov y,r0
    1:
```

This is written:

```
if (r0 = x < y) r0 = y;
```

The *relational operator* < specifies what *condition* you wish to test for. The compiler will generate a cmp or tst instruction, if necessary, and figure out the best way of branching to achieve the desired flow of control. If you want to compare 16-bit unsigned addresses for 'lower' or 'higher,' use << or >>, respectively, instead of < or >. Equality testing is specified by ==, and inequality by ~=.

If you have reason to believe that the condition code is already properly set, you can just write >=, for example, to specify the condition 'greater than or equal to.' There are also keywords for each of the four condition code bits — **minus**, **zero**, **oflow**, and **carry**. In conjunction with the not operator ~ it is possible to specify any machine defined condition:

```
r0 = x + y;
if ( < ) r0 =- r0;
if (~oflow) z = r0;
else z = 32767;
```

The =- operator puts the negative of r0 in r0, i.e. neg r0.

3. Boolean Connectives

Tests are often much more complicated. An alphabetic symbol in LIL, for instance, is any upper or lower case letter, dot, or underscore. To set r0 to 1 if it contains an alphabetic character, and 0 otherwise:

```
        cmp r0,'a       / is it between 'a'
        blt 1f
        cmp r0,'z       / and 'z'
        ble 2f
    1:  cmp r0,'A       / or between 'A'
        ble 1f
        cmp r0,'Z       / and 'Z'
        ble 2f
    1:  cmp r0,'.       / or is it '.'
        beq 2f
        cmp r0,'_       / or '_'
        bne 1f
    2:  mov $1,r0       / 1 to r0 if so
        br 2f
```

```
1:      clr r0          / 0 otherwise
2:
```

In LIL this becomes

```
if ('a' <= r0 && r0 <= 'z'
        || 'A' <= r0 && r0 <= 'Z'
        || r0 == '.'
        || r0 == '_')    r0 = 1;
else r0 = 0;
```

The operators && and || let you combine conditionals in a fairly obvious fashion. If the left condition before a && is false, the combination must be false, so the right condition is skipped. Similarly, a true condition before || lets you skip evaluation of the following condition. If the right condition must be evaluated, then it determines the truth of the combination. && binds tighter than ||, the way we are accustomed to reading logical statements. Again the tests are performed, as needed, strictly left to right.

You can also use parentheses and the ~ operator to alter the normal binding order or invert the meaning of any part of a test.

Remember that *two* equal signs in a row are used to specify that you are testing for equality, because one equal sign alone always means 'assign.' Had the test ended as

```
        || r0 = '_')      % WRONG!!
```

it would have meant, 'or, copy the character _ into r0 and if r0 is subsequently nonzero, the test is true.' This clobbers r0 and makes the test always come out true — so watch out. (That % marks a comment, by the way. Everything from the % through the end of the line is ignored by the compiler.)

You might also observe that we wrote

```
r0 = 0;
```

where we wanted a clr r0 instruction. You can trust the compiler to nearly always generate the best instruction for the job, so there is no special notation for clr. Thus, should you change some parameter to zero and recompile, clrs will appear wherever possible. Similarly inc, dec, and quite a few other special purpose instructions are generated automatically. The tabulation of operators shows when they are used.

4. Else

As the last example showed, you can always follow an if by an else, which specifies what action to take if the test is false. The else part is skipped over when the test is true.

An important use of the else clause is to string together a series of tests to make what is known as a 'case switch' in some languages. This is a multi-way if that does at most one of several different cases. LIL permits you to write certain character constants (between single quotes), for instance, using 'escape' sequences to make invisible characters apparent. \t becomes a tab character, \n a newline. The code to perform this conversion looks something like:

```
if      (r0 == '0') r0 = 000;
else if (r0 == 'a') r0 = 006;
else if (r0 == 'e') r0 = 004;
else if (r0 == 'n') r0 = 012;
else if (r0 == 'p') r0 = 033;
else if (r0 == 'r') r0 = 015;
else if (r0 == 't') r0 = 011;
else ;              % default is \x = x; or r0 = r0;
```

The first test that succeeds, reading down from the top, causes the corresponding action to be performed; execution then resumes following the last else. In this case, the last else has no test and so becomes the *default* action — what to do if none of the tests succeeds. Since the default action here is the null statement ';' the else could have been left off. It was written as a reminder, however, which is often a good idea. No harm is done because the compiler is smart enough not to generate extraneous code.

Those numbers that begin with 0 in the example above are octal.

Since the statement controlled by the if can be anything, including another if, a natural ambiguity arises when the else option is used. When you say

```
if (a)   if (b) ...;
else ...;
```

to which if does the else belong? The compiler arbitrarily resolves the matter by matching the else to the second if. This is not what the indentation implies! To get the flow of control correct, you would have to add a null else:

```
if (a)   if (b) ...;
         else ;
else ...;
```

There is no need for this rather ugly form, however, since you can more easily write

```
if (a && b) ...;
else ...;
```

and get the desired effect. If you really want the other binding of the else (which is also an ugly way of coding), at least make the if-else into a group.

5. Groups

Just as you can group parts of an expression with parentheses, so too can you group statements together. Use braces:

```
if (x < y)
        {r0 = x;        % exchange x and y
        x = y;
        y = r0; }
```

Now the if controls the entire group instead of just one of the statements. Groups may appear anywhere you see an expression ending in a semicolon (which is one of the simplest statements allowed), including inside other groups. Note that the closing brace occurs *after* the last semicolon (and no semicolon is written after the braces). Semicolons *end* things, they don't separate them.

6. Loops

Loops are another important type of control structure. They often control a group of statements:

```
r0 = 0;
while (r0 < nitems)
        {a[r0] = b[r0] - c[r0] + 1;
        r0 + 2; }
```

The square brackets indicate indexing, with respect to r0. In this form of the loop statement, the test is made at the top. That way, if nitems happens to be zero, the body of the loop is safely skipped, which is usually what we want to happen. If the test is met, the statement following the test, which happens to be a group, is executed and control returns to the test. The test will

eventually fail, in this case, and execution will continue with the statement following the loop.

Loops can also take another form:

```
r0 = 0;
do {    area1[r0] = 0;
        area2[r0] = 0;
        } while (r0 + 2 < 500);
```

This clears 250 elements of the arrays area1 and area2. Since we know that the loop will be obeyed at least once, we can put the test at the end and get a tiny reduction in the number of branches required. More important, we can state the increment and test a little more compactly.

The do-while form can also be followed by a non-null statement (or group, of course) instead of just a semicolon:

```
r0 = listhead;
r1 = 0;
do      r1 + 1;
        while (next[r0])
        r0 = next[r0];
                        % r0 points at last element
```

It is rare that you need a loop with the test somewhere in the middle — in fact you should make sure you really do before writing it that way — but the facility is there if you need it.

7. Break, continue, and goto

The break statment lets you exit from a loop at other places besides the test:

```
r0 = 0;
do      if (a[r0] ~= b[r0]) break;
        while (r0 + 2 < 500);
```

As soon as corresponding elements of the arrays a and b are found that do not match, the loop is exited. A loop may contain any number of breaks.

Sometimes you want to go back to the top of a loop from several places. Use continue:

```
while (true)
        {_getchar();
        if (r0 == ' ' || r0 == '\t' || r0 == '\n')
                continue;
        ...; }
```

The keyword true specifies a condition that is always met. It is used in this case to form an 'infinite' loop with no tests. _getchar() is a subroutine call — more on subroutines later (and the reason for that silly _) — to a routine that returns an input character in r0. Thus this loop header skips all blanks, tabs, and newlines on input.

You should be able to specify nearly all your control flow with if, else, do, while, break, and continue. If you make a habit of doing so, in fact, you will find your code much easier to read, and less likely to contain bugs. Occasionally, however, you may find a need for more direct control. The goto statement lets you specify an arbitrary br or jmp instruction:

```
goto jail;
goto brtable[r0];       % table of branches
goto [atable[r0]];      % table of branch addresses
```

As the last form shows, indirection is specified by square brackets in a way analogous to indexing.

8. Addressing

We have seen a number of different kinds of operands, implying different modes of addressing. Now is a good time to complete the list.

Every operand (symbol, literal, expression) in LIL has a number of *attributes* besides value. One of the most important is its *type*, which corresponds very closely to legitimate addressing modes on the PDP-11. A constant such as 3, for instance is assumed to be an *immediate* reference to a literal 3. If you have to refer to absolute *memory* location 3, write **mem** 3. Similarly, **reg** 3 is equivalent to the predefined symbol **r3**, or *register* 3.

On the other hand, you sometimes want to get hold of the address of a memory reference, instead of its contents. **&x** has the same value as **x**, but is of type immediate:

```
r0 = &x; r1 = 0;
while ([r0]++) r1 + 1;
```

This counts the number of non-zero entries in the array **x**, up to but not including the first zero. The notation **[r0]++** specifies *autoincrement* addressing; The **++** comes after the brackets to remind you that incrementing occurs *after* the reference is used. It is the only operator that follows operands.

To put **r0**-**r5** on the stack:

```
--[sp] = r0 = r1 = r2 = r3 = r4 = r5;
```

Since the first operand remains unchanged, *autodecrement* occurs on every assignment.

As we saw earlier, square brackets indicate indexing, as in **x[r0]**, indirection, **[x]**, or both, **[x[r0]]**. Any valid PDP-11 addressing mode is acceptable, even **[[r0]]** (which becomes **[0[r0]]**). And, of course, expressions are allowed inside brackets:

```
a[r0 = i] = b[r1 = j + 2];
```

9. Compile-time expressions

Most assemblers allow you to do a limited amount of arithmetic with symbols. To move the contents of location **b+2** into **r0**, for instance, you just write

```
mov b+2,r0
```

LIL uses operators like **+** and **−**, however, to specify machine instructions. So a special notation is used to show which computations are to be done at *compile-time:*

```
r0 = "b + 2";
```

Expressions inside double quotes generate no machine instructions, but are computed directly from the values of symbols and literals. You can look on compile-time expressions as instructions to a special machine whose storage locations are the symbol table entries. Most of the run-time operators have analogous compile-time meanings.

Compile-time expressions have a slightly different interpretation than the run-time expressions used to specify PDP-11 code. We want

```
b + 2 -> r0;
```

to add 2 to location **b**, then move the contents of **b** into **r0**. But

```
"b + 2" -> r0;
```

does not redefine the symbol **b** as having a value bigger by 2 than before — the expression simply *refers* to the value of **b** and leaves the symbol alone, which is what we usually want. So this form, as before, moves the contents of location **b+2** into **r0**.

To *define* a symbol, i.e. actually set its value, use the compile-time assignment:

```
"x = b + 2";
```

This defines x initially as having the value (and other attributes of) b, adds 2 to that, then closes x for any further redefinition, with the end of the compile-time expression. Hereafter we can write

```
r0 = x;
```

to move the contents of b+2 into r0.

Type and other attributes are also set in compile-time assignments:

```
"isp = [sp]"; "sink = --isp";
sink = r0 = r1 = r2 = r3 = r4 = r5;
```

As you can see, addressing modes can be built up in stages.

10. Labeled Groups

The usual way to define a symbol is with a *labeled group:*

```
x{ ...; }
```

This notation causes several things to happen:

1) The code between braces is a *local region* — symbols may be defined inside it with the same names as those in other regions, or even in containing regions, without conflict. Local definitions disappear after the close of the group.

2) **break** and **continue** statements may be used to skip to the bottom or top, respectively, of the group, just as with a loop. If the statement has a label, as in **break x**, the **break** (or **continue**) ignores all containing loops and labeled groups whose labels do not match the label in the statement, and instead applies to the innermost containing group whose label matches.

3) The label x is defined (in the containing region, *not* just local to the group) as being a memory reference whose value is the location counter at the start of the group, and whose *size* is the number of bytes of code inside the group (more on both the location counter and the **size** attribute later).

Labeled groups are used as the bodies of subroutines, or of important subunits of routines, or just to define data:

```
array{1; 1; 2; 5; 8; 11; }
scalar{-1; }
label{; }                    % just define label
```

Note that you can generate constants in line simply by writing them. (This is how you sneak in special instructions like **wait** and **iot**.)

When used as a local region, it is best to head the group with a **local** statement:

```
x{      "local a, b, c";
        ...; }
```

This declares a, b, and c to be symbols that will be defined inside the group, and that should be forgotten on exit from the group.

If not explicitly declared, a symbol is made local to the first group in which it is referenced. Should the group end before it is defined, the symbol is promoted into the next outer containing group. Any symbols undefined at the end of a program file are published as *undefined externals.*

11. Subroutines

No special code is generated at either the start or the end of a labeled group, even one used as a subroutine body. Thus there are no preconceived notions about how you must build subroutines, which is by way of saying that you are obliged to write entry and exit sequences explicitly.

For convenience, however, there *is* a presumptuous mechanism for *calling* routines. Designed to be compatible with code produced by and for the language C, the two ways of calling a routine are:

```
fun();              % no arguments
fun2(arg1, arg2, r0 = arg3, arg4 + 1);
```

The code generated is

```
jsr pc,fun          / first call
mov arg3,r0         / arg expressions
inc arg4            /  left-to-right
mov arg4,-(sp)      / args moved
mov r0,-(sp)        /  onto stack
mov arg2,-(sp)      /   right-to-left
mov arg1,-(sp)
jsr, pc,fun2        / second call
add $8,sp           / args popped
```

The arguments are moved onto the stack in *reverse* order so that 2[sp] always refers to the first argument, 4[sp] the second, and so forth. That way, if the wrong number of arguments are passed, the first ones still match up properly and no major harm is done.

To *receive* this sequence takes no special effort, although you might wish to use the C library routine

```
jsr r5; rsave; n;
```

to save nonvolatile r2-r5, reserve n bytes on the stack, and leave r5 pointing at the argument list (so arg1 is at 4[r5], arg2 at 6[r5], etc.). You would then leave, after putting any integer result in r0, by

```
goto rretrn;
```

That jsr r5 is no mistake — LIL treats jsr as a unary operator, similar to - or ~, which changes a register into an immediate that looks like the first word of a

```
jsr r5,*$subr
```

instruction. To return from a subroutine, write (you guessed it)

```
rts pc;
```

And, to complete the record, you can make system calls with a

```
sys n;
```

where n is an immediate.

For added convenience, LIL treats C compatible function calls as references to register 0:

```
if (alf(c)) x = c;
```

becomes

```
mov c,-(sp)
jsr pc,alf
tst (sp)+           % sneaky form of sp+2
```

```
        tst r0
        be  1f
        mov c,x
1:
```

And if you plan to use a large number of these function calls, you may wish to do as the C compiler does — reserve a temporary location of top of the stack, [sp], to hold the last argument on all calls. Thus

```
fun(c)
```

becomes

```
mov c,*sp
jsr pc,fun
```

in general saving an autodecrement and an add on each call with arguments. To turn on this feature, declare a local version of the flag symbol .temp and set it nonzero.

12. More declarations

If you want your function names, or any others, to be used to satisfy references in other compilations, use

```
"extern a, b, c";
```

extern is an additive attribute — it can apply to symbols also declared local. In fact, if you wish to refer only in some sub-region to an undefined external, you must declare it in both local and extern statements inside that region.

In case you have to use any of the LIL keywords to communicate with, say, assembler routines, you can turn off their special meaning, once and for all, with an escape:

```
"extern \if";
"xif = if";
```

This is best done, as shown, at the end of a program file, to define the alias used in the body of the code.

A useful variant of the local statement looks like a compile-time function call:

```
"0[sp](a, b, c sizeof 4, d, e)";
```

The effect of this statement, for each argument from left to right, is to

1) declare the argument to be local and define it to be equal to the label (originally 0[sp]).
2) increment the value of the label by the size of the argument (e.g. 0[sp] becomes 2[sp]). Default size for all symbols is 2, but the sizeof operator can be used to specify another size.

If used as an operand, this form thus has a value equal to its original value plus the size of everything declared, in this case 12 (since the sizeof operator explicitly declared c to be of size 4).

This form is most useful in defining names for *automatic* variables (reserved on the stack every time a routine is entered), in conjunction with the C entry sequence

```
subr{   jsr r5; rsave; sizeof "0[sp](a, b, c sizeof 4, d, e)";
```

As a unary operator, sizeof changes its operand into an immediate reference whose value is the size of the operand, so the proper number of bytes to be reserved will be generated in line.

Another convenient notation resembles indexing:

```
a[2] = r0;      % is the same as
"a + (2 * sizeof a)" = r0;
```

where the * specifies compile-time multiplication.

13. Load control

Sizes are not used by LIL to generate code — mostly the compiler keeps track of them for handy reference, as above. If a symbol remains undefined, however, its size is published along with its name as an undefined external. The UNIX loader treats such a reference with a non-zero size as a *labeled common block*. All references with the same name are collected by the loader (keeping track of the maximum size requested), so that sufficient space can be allocated in the *bss* section.

Generally, you compile code into the *text* section, compile initialized data into the *data* section, and just layout data areas in the *bss* section (*bss* is a fossil meaning 'block started by symbol' in middle low Phoenecian). UNIX promises to zero everything in the *bss* area, but won't let you help. LIL therefore will smack your fingers if you try to generate code except into the *text* or *data* areas.

How do you switch loading among areas? There is a special symbol called dot '.' which is set to the current value of the location counter at the beginning of every statement. You may redefine dot yourself, so long as you don't try to back up over code already generated. It is generally meaningful to set dot only to one of the other redefinable symbols: **.abs**, **.text**, **.data**, or **.bss**. These are the location counters for absolute (unrelocated) loading, or for one of the relocatable sections already discussed. Thus:

```
". = .data"{                    % switch to data area
        array{1; 1; 2; 5; 8; 11; }
        scalar{-1; }
        label{; }               % all defined in data area
              }                 % switch back
```

Using dot to label a group as shown provides for a temporary diversion of loading, since the label is defined at the close of the group as having the value of the location counter at the start of the group. If you want to generate a lot of code into, say, the *data* area, use the simple switch:

```
".=.data";
```

It is not necessary to switch dot unless you plan to generate code (although dot can be made to point in many interesting directions). You can use the shorthand

```
".bss(a, b, c sizeof 4, d, e)";
```

as before, only this time the location counter for the *bss* area is updated to make room for the variables. Code before and after this statement will go into the *text* section, as usual, unless otherwise specified.

It is good practice to keep data that must be altered out of the *text* section, by the way. That way, you can often write-protect the code (hardware permitting) and make your debugging job much easier.

14. Bytes and strings

We have come a long way without mentioning byte addressing. There is nothing terribly magic about it — just use the **byte** modifier before any operand you wish to be used as a byte reference.

```
r0 = byte s;
byte d = byte s;
byte d = '\0';
```

The rule is — if either operand is of type byte, the other must also be of type byte, unless it is an immediate or a register.

If you get tired of writing byte, make it part of the symbol definition:

```
".bss(byte name sizeof 8)";
r0 = 4[r5];
r1 = 0;
do      name[r1] = byte [r0]++;
        while (r1 + 1 < sizeof name);
name[r1] = '\0';
```

And if you must 'undo' the byte attribute, use word.

Single quotes are used in LIL to denote character strings of any length. Odd length strings are padded to even length by adding a null byte, but otherwise no effort is made to ensure a null terminator. Strings up to length two fit in one word and may be used wherever an immediate is allowed. Longer strings may be used only to generate ascii text in line:

```
table{  0; 'zero';
        1; 'one ';
        2; 'two '; }
```

or as a literal reference:

```
if (debug) _printf(='error message\n');
```

The unary operator = compiles the operand string (or any immediate) into the *data* section. If the odd byte of the last word is not zero, a zero word is appended. The operand becomes an immediate whose value is the address of the first word allocated. Thus, the literal notation above is equivalent to the C string reference "error message\n".

15. C compatibility

Most of the job of interfacing to the C world is handled in the subroutine call/return machanism already described. You should be familiar with C and its call by value convention to intermix code safely. C insists on prepending a _ to all symbols referenced in that language. This avoids conflicts with assembly language support routines, but makes for generally unpretty names in LIL.

If you run under the aegis of the C support library, you must provide a main routine, called _main, and should leave gracefully by returning from _main or by calling _exit.

Be warned that

```
_fun(byte a);
```

will move only one byte onto the stack, leaving garbage in the high-order byte. C expects a properly sign-extended integer.

Also, thanks to the curious way that things are declared, you must be careful with undefined external function names. If the size of such a reference is non-zero (and the default is 2), the UNIX loader will not match the reference up to a text symbol. LIL sets the size attribute to zero on any symbol used as the entry name on a C compatible call. But for nonstandard usage, you must say

```
"extern rsave(), rretrn()";
```

to set entry name sizes to zero.

Most of all, if you mix languages, *watch out for the differences.*

16. Preprocessor

LIL uses a preprocessor essentially identical to the one provided with C. If the first character of a program file is #, the file is scanned for lines that look like

```
#include 'pathname'    /* the specified file is included */

#define SYMBOL string      /* all subsequent occurrences of
                           SYMBOL are replaced by the
                           defining string, surrounded by
                           blanks */
```

PL/I style comments are used only in conjunction with preprocessor statements.

Some useful things to define are:

```
#define ENTER       jsr r5; rsave;
#define LEAVE       goto rretrn
#define RETURN      rts pc
#define AUTO 0[sp]
#define ARGS 4[r5]
```

If you have more than one case at your disposal, it is a good idea to write defined symbols in all caps to distinguish them from other symbols.

17. Conditional compilation

LIL also provides for *conditional compilation*, with the aid of *compile-time conditions* and the ordinary conditional statements of the language. Compile-time conditions must evaluate to strictly true or false, and may be combined with those keywords to determine whether a piece of code should be omitted:

```
if ("recsize == 512") getblock();
else read(fcb, &buffer, recsize);
```

Depending on the value of the symbol recsize (which must be known), only one of the two function calls will be compiled into the final program. The else is, of course, optional:

```
if (debug)
        {dcount + 1;
        _printf('got this far %d\n', dcount); }
```

This can be turned on or off simply by writing

```
"debug = true";
```

or

```
"debug = false";
```

or it can be made a runtime switch by writing

```
".bss(debug)";
```

and setting it on the basis of some input character at run-time.

Loops can also be turned on or off, from the test onward:

```
do      ...;            % this is compiled
        while (false)
        {...;           % this is skipped
        }               % no jump back
```

for what it may be worth. A useful form, however, is the 'infinite' loop we saw earlier.

18. Syntax, character set

By now you probably have a pretty good feel for the syntax of the language, without even being told the nuts and bolts rules. Blanks, tabs, and newlines are largely ignored (but a string may not contain a newline — you must use the escape sequence \n). It helps if you are careful to indent to show nesting depth, as the examples have showed. And you may write names of any length, for clarity, but only the first eight characters are meaningful.

Operators may *not* be run together as in most languages:

```
x ->--[spl;  % is illegal
x -> --[spl; % is required
```

It is possible to type LIL programs quite readably using just the basic ascii 64-character set. Braces can be replaced with ordinary parentheses (since LIL doesn't distinguish them anyway), ^ for |, and ! for ~. The compiler does distinguish between upper and lower case letters, so beware of trying to alter a mixed-case program from a one-case terminal. That way madness lies.

19. Compiling

To use the LIL compiler, type the UNIX command line

lc args

where args are source files if they end in .l, flags if they begin with —, and object files otherwise. Flags and loading conventions are just like those for the C compiler (where appropriate see cc(I) in the UNIX manual). Some important flags are:

—p Include 'profile' library, which will automatically produce a histogram of code usage when program is run (see prof(I) in UNIX manual).

—P Preprocess only, writing expanded source on file with .l changed to .i.

—c Compile only, writing standard UNIX object code on file with .l changed to .o.

Much of this baggage is for use with C world programs. If you are coding entirely in LIL, say for another PDP-11, you will probably just use

/usr/pjp/pdp/lc —c files

where each of the files is a source file whose name ends in .l. In this case, each file is compiled in turn and its object code written on a file with the same name, only ending in .o. You can then bind object decks together with the loader (see ld(I)), to make a standard UNIX executable file. This is certainly the best format to work from, since it lets you mix in existing assembly language code, and anything else that UNIX provides (even Fortran).

20. Unary operators

As promised, we now present a complete list of all the operators that LIL recognizes, starting with the unary (one argument) operators. These have the same meaning at compile-time as at run-time, for none of them generate code. Instead, they modify *references* to symbols or expressions, by changing their type, value, or other attributes. For convenience, we define the following symbols:

l non-byte memory reference.

m, n absolute (i.e. unrelocatable) immediate.

r register.

v, w any non-byte address type.

x, y any PDP-11 address type.

The unary operators are:

x++ converts type register indirect to type autoincrement.

--x converts type register indirect to type autodecrement.

=n compiles the absolute immediate or string n into the *data* section, appending a zero word if the last byte is not null. Result is type immediate, value is address of start of the allocated string.

&x converts x to type immediate, removes byte attribute.

mem x converts x to type memory, removes byte attribute.

reg x converts x to type register, removes byte attribute. x must be defined absolute with 0 <= value <= 7.

word x removes byte attribute.

byte x attaches byte attribute.

-n negates absolute immediate n.

~n complements absolute immediate n.

rts r converts type register r to an immediate with value of rts r instruction.

jsr r converts register r to an immediate with value of jsr r,*$ instruction. (Should be followed by address of routine.)

sys n converts immediate n to an immediate with value of sys n instruction.

sizeof x converts x reference to an absolute immediate with value equal to size attribute of x reference.

||| n shifts absolute immediate n left 8.

21. Compile-time binary operators

x>=y, x>y, x==y, x<y, x<=y, x~=y result is a condition with value true if the relation obtains, otherwise false. Both x and y must have the same relocation bias. Comparisons are all on signed quantities.

x=y, y->x x is defined to have the same type, value, and other attributes as the y reference. x must be previously undefined (unless it is dot or one of the four location counters).

x=-n x is defined as the negative of the absolute symbol n.

x=~n x is defined as the complement of the absolute symbol n.

The type of x is left unchanged in the following:

x+y result is the sum of x and y. At most one of the two may be relocatable or undefined.

x-y result is the difference of x and y. If y is relocatable or undefined, it must have the same bias as x.

n~m result is n equivalence m (not exclusive or).

n~~m result is n exclusive or m (not equivalence).

n|m inclusive or.

n&m and.

n&~m same as n&(~m).

n**m n is shifted left (if m>0) or right (if m<0) by |m|.

n*m multiplication.

n/m division.

n|||m same as (n&0377) | (m**8) [packs bytes].

x sizeof n result is x with size n.

22. Run-time binary operators

These are the creatures that actually generate code. If x and y are not both byte or both word, then one of them must be a non-byte register or immediate. The instruction generated will then be byte mode if either operand is byte. The following is a metalinguistic description of the decisions made by LIL in producing code for each operator.

```
x>=y, x>y, x==y, x<y, x<=y, x~=y        [compare signed]
x>>=y, x>>y, x<<y, x<<=y                [compare unsigned]
        if (y==0 && cc set on x)        do nothing
        else if (y==0)                  tst(b) x
        else if (x==0)                  tst(b) y
        else                            cmp(b) x,y

x=y,
y->x    if (y==0)                       clr(b) x
        else if (x=={carry oflow zero minus} && y=={true false})
                                        setx or secx
        else if (y==minus)              sxt x
        else                            mov(b) y,x

x=-y    if (y==0)                       clr(b) x
        else if (x==y)                  neg(b) x

x=~y    if (x=={carry oflow zero minus} && y=={true false})
                                        secx or setx
        else if (x==y)                  com(b) x

x+y     if (y==1)                       inc(b) x
        else if (y==carry)              adc(b) x
        else                            add y,x

x-y     if (y==1)                       dec(b) x
        else if (y==carry)              sbc(b) x
        else                            sub y,x

x|y                                     bis(b) y,x

x&n                                     bic(b) $!n,x

x&~y                                    bic(b) y,x

x^r                                     xor r,x

r*y                                     mul y,r

r/y                                     div y,r
```

```
x?y     if (y==0)           tst(b) x
        else                cmp(b) x,y

x?&y                        bit(b) x,y

x<>n    if (n==1)           rol(b) x
        else if (n== -1)    ror(b) x

x<*>n   if (n==8)           swab x
        else if (x odd reg) ashc n,x

x**n    if (n==1)           asl(b) x
        else if (n== -1)    asr(b) x
        else if (x a register) ash n,x

r***n   ashc n,r
```

Both the sizeof and lll operators are also defined at run-time and have the same effect as at compile-time. They do not directly cause code to be generated.

The compiler is aware of what is happening to the condition code most of the time. It knows, for instance, that swab and function calls do not leave the code in a state implied by the language, and so will generate a tst if the result of either is to be used in a test. The PDP-11 is somewhat whimsical about the setting of the carry bit, however, and LIL makes no real attempt to second guess the machine. (It makes a difference whether you add 1 or 2 to something, for instance.) It is always a good idea to be very careful when testing for special conditions.

23. Machine instructions

It is best to learn to think in terms of LIL statements, instead of individual machine instructions. But to help you make the conversion, here is a list of the PDP-11 orders and how to generate them one at a time. Symbols are the same as in the previous sections.

```
adc(b) x        x + carry;
add v,w         w + v;
ash r,n         r ** n;                 % |n| ~= 1
ashc r,n        r *** n;
asl(b) x        x ** 1;
asr(b) x        x ** -1;
bcc l           if (~carry) goto l;
bcs l           if (carry) goto l;
beq l           if ( == ) goto l;
bge l           if ( >= ) goto l;
bgt l           if ( > ) goto l;
bhi l           if ( >> ) goto l;
bhis l          if ( >>= ) goto l;
bic(b) x,y      y &~ x;
bis(b) x,y      y | x;
bit(b) x,y      y ?& x;
ble l           if ( <= ) goto l;
blo l           if ( << ) goto l;
blos l          if ( <<= ) goto l;
blt l           if ( < ) goto l;
bmi l           if (minus) goto l;
bne l           if ( ~= ) goto l;
```

```
bpl l          if (~minus) goto l;
bpt            3;
br l           goto l;
bvc l          if (~oflow) goto l;
bvs l          if (oflow) goto l;
ccc            0257;
clc            carry = false;
cln            minus = false;
clr(b) x       x = 0;
clv            oflow = false;
clz            zero = false;
cmp(b) x,y     x ? y;                    % note inversion
com(b) x       x = ~ x;
dec(b) x       x - 1;
div w,r        w / r;
emt n          "014000 + n";
halt           0;
inc(b) x       x + 1;
iot            4;
jmp w          goto w;
jsr pc,w       w0;
jsr r,*$l      jsr r; l;
jsr r,w                                  % not provided
mark n         "06400 + n";
mfpi w                                   % not provided
mov(b) x,y     y = x;
mtpi w                                   % not provided
mul w,r        r * w;
neg(b) x       x = - x;
reset          5;
rol(b) x       x <> 1;
ror(b) x       x <> -1;
rti            2;
rts r          rts r;
rtt            6;
sbc(b) x       x - carry;
scc            0277;
sec            carry = true;
sen            minus = true;
sev            oflow = true;
sez            zero = true;
sob l                                    % not provided
sub v,w        w - v;
swab w         w <*> 8;
sxt w          w = minus;
trap n         sys n;
tst(b) x       x ? 0;
wait           1;
xor r,w        w ~~ r;
```

24. Keywords and predefined variables

Keywords are reserved words in LIL, which means you cannot have local variables with the same names as keywords (unless you turn off their special meaning with the escape \, as we explained earlier). Here is a complete list of the keywords:

break	**goto**	**rts**
byte	**if**	**sizeof**
continue	**jsr**	**sys**
do	**local**	**while**
else	**mem**	**word**
extern	**reg**	

Predefined variables, on the other hand, may be superseded by new local definitions. They are merely provided for convenience or to make available special services which you may not need:

.	**false**	**r3**
.abs	**minus**	**r4**
.bss	**oflow**	**r5**
.data	**pc**	**sp**
.temp	**r0**	**true**
.text	**r1**	**zero**
carry	**r2**	

25. A final note

Performing compile-time arithmetic with the current location counter, dot or '.', is generally unsafe, since it is difficult to anticipate how many words will be needed to represent a given sequence of instructions. LIL may actually evaluate the same expression differently in different places as the compiler learns more about what instructions may be made shorter. Since dot is known to be set at the beginning of each statement, there are occasions when it is both safe and useful to write compile-time expressions involving dot. One important situation is the indexed branch used to implement an efficient 'case switch':

```
if (MIN <= r0 && r0 <= MAX)
        switch {
        goto ". + 4 - MIN"[r0];
        "local case0, case1, case2, ...";
        case0; case1; case2; ...;

case0{  ...; }
case1{  ...; }
case2{  ...; }
        ...;
                }       % end of switch
else    ...;            % default action
```

Here MIN and MAX are manifest constants specifying the valid limits of the branch index (which of course must be even). The group labelled switch permits the case labels to be kept local to this particular switch, so they can be used repeatedly. Default actions, if any, are provided by the else clause.

P J Plauger

MH-1352-PJP

P.J. PLAUGER

20