LITTLE Newsletter # 39

T. Stuart
September 15, 1975

Post-partum reflections on the Honeywell minicomputer implementation of LITTLE. Minicomputer Software

Dismay is the most likely response for a programmer confronted with developing substantial systems on a minicomputer(1). If any language other than the machine language exists, it is most likely to be FORTRAN or BASIC at which point the manufacturer's software budget ran out. And if they are available, they surely will not be optimizing compilers. In a few instances a machine oriented language is available, for example PL-11 and BLISS-11 on the PDP-11, PL516 on the Honeywell Series 16, and ALIAS on the PDP-9. All of these languages are improvements over the alternative assembly code for systems programming. They all provide access to machine characteristics and produce efficient code and should have received wider use. The efficiency is not the result of any optimization techniques but rather the low level and direct references to registers or addresses and imbedded assembly code. The size of minicomputers generally dictates the exclusion of very much optimization in any case, so a higher level language in which optimization becomes imperative is not practical. For a general discussion of these minicomputer systems implementation languages and several others for larger systems, see Reference 2.

Faced with implementing a graphics system of some size on a Honeywell Series 16 computer, and having previously worked in a combination of FORTRAN and an assembly language for the same purposes, there was every reason to consider a systems development language. An additional criterion was machine independence, needed partly because some graphics work would proceed on the minicomputer, while others would run on a larger system, and still others on both, and partly because other minicomputers might eventually be used.

Choosing LITTLE

The choice to implement LITTLE was made primarily on the basis of its design around the twin goals of machine independence and efficiency. The resolution of the conflict between these somewhat contradictory goals through design decisions involving the language syntax and the compiler structure has been described earlier. Beyond these assets there were some characteristics of special note for this implementation. First, the presence of macros within LITTLE had a benefit other than the obvious increase in the language's power. Specifically, in graphics the capability for expressing command or drawing instructions in a natural, more English-like manner was a desirable feature; macros enable this approach. Second, the size of the LITTLE compiler is much larger than any minicomputer will support. Thus, it runs as a cross-compiler. Perhaps, this would seem a disadvantage, but not so. Since LITTLE is machine independent, program development and debugging take place on the larger machine with its extensive aids. Third is the inclusion of most of the FORTRAN syntax in LITTLE. Though code written by experienced LITTLE programmers generally looks more like ALGOL, programmers suffering from an exclusive FORTRAN background can express themselves without difficulty.

Few systems implementation languages can support a claim for machine independence. Of those with independent qualities none are both efficient and possess these added characteristics. On the other hand probably none are as difficult to implement well as LITTLE is.

Target Machine

The Honeywell series 16 computers are characterized quickly as sectorized, accumulator,twos-complement machines. The two useful registers are the accumulator and the index register. Machine instructions are all one word in length and have a single operand. For instructions that reference memory, the operand field occupies nine bits, thus limiting references to 512 words, the size of sectors. The references are absolute, not relative to the instruction. One bit of the instruction allows switching the reference to the first sector of memory, sector zero. References to locations outside both sector zero and the sector of the instruction must be accessed through indexing or indirect addressing or both. When both are used there is a complication. It arises because the original version of the machine caused indexing to be specified on the indirect address rather than the final direct address. This design fault was corrected by a hardware option that allows one to switch addressing modes. Not all machines possess it. The machine instruction repertory is rather standard. Multiplication and division are options absent from many configurations.

The only software provided by the manufacturer of interest to this project was the FORTRAN compiler, the assembler, and the loader. FORTRAN was of interest mainly in order to define the calling convention for routines; except for four short library routines nothing of that language has been used. Two versions of the assembler were available, the standard one and another which ran as a cross assembler on the CDC 6600. The latter was used, and of its use, more later. The loader used was the most complex of those available, but was still primitive. The most serious defect was a lack of provision for data blocks of global variables. Though a "common" existed for the loader, its properties bore no relation to a usual block of this type.

Given the machine characteristics, one can see a number of problems:

1. Compilation must handle two addressing modes. (Mode independent code is possible but inefficient.) Execution requires two run-time libraries.

2. Operands of machine instructions preferably will not refer to addresses in other sectors, because this costs space and execution time.

3. Effective use of sector zero will mean placing the most frequently used quantities there.

4. Some convention must be devised for linking global data blocks.

These problems are unique for this implementation, but every machine will have an equivalent set to superimpose on the basic task of code generation.

Design Decisions

The first choice to be made was the number of LITTLE primitives to implement. Since there were expectations of a substantial need for machine independent code right from the beginning of the compiler's use, a larger set of primitives, forty, were chosen than is necessary for a useful compiler. The only primitives omitted were real number operators and functions. Most systems work has no need for real numbers, and in the exception, FORTRAN routines could take care of the problem. Two score primitives would not constitute a major task for implementation in many languages, but because the primitives output by the parser are still devoid of any features of a machine dependent nature, and because the goal of efficiency must be attained largely through the work of the code generator, the task in LITTLE is major.

The second choice was the form of the output, binary or symbolic. Proceeding directly to binary output and bypassing an assembler has two distinct advantages. It is more efficient in terms of execution time for the compilation process, and it avoids the idiosyncracies of an assembler, expecially the limitations on symbols. On the other hand, employing an assembler also has a couple of advantages. The assembler will perform a service by checking for errors in the translation. And the presence of assembly code will then enable modifications or extensions on the machine level to handle operations not available in the higher level source code. Symbolic code emission was the choice, and as it turned out, the first reason for choosing it was the major justification. There has never yet been any code tinkering; when a hardware device must be addressed, an assembly language routine is written. However, a third reason did surface; another installation desiring some software, but without a LITTLE compiler, was supplied with the assembly source.

Another major problem involved communication between object modules. As noted earlier, the loader provided no means of linking global data blocks. To be precise, the problem is to enable reference to global variables from assembly language programs. One solution that would have been easy on the LITTLE program writer is to adopt a convention for naming these blocks, and then passing them off to the assembler and loader as subroutines. There were several disadvantages to this. First, the assembler and the LITTLE compiler accept the same set of characters as legal; hence no special character is available that would be an acceptable convention for linking to assembly programs. The unacceptable alternative is to impose restrictions on symbol use in LITTLE source code. Furthermore, referencing an element of an array that is in turn a member of a global block would be a cumbersome and error prone task in assembly language.

Yet another aspect of the problem is that the Series 16 machines have dedicated machine locations that it would be advantageous to address from LITTLE. Since we are writing systems programs, the lack of a capability to tie locations to variables causes a resort to assembly programming. Treating global blocks as external, relocatable routines aids not at all.

It is even debatable whether there is always an advantage in treating a global block that appears as an entity in source code as an entity in object code; storage allocation can be more efficient when the variables are treated separately, and this is discussed later.

Considering the above, two realistic solutions appeared; one was to re-write the loader, to change our universe as it were, and the other was to establish a procedure for specifying machine addresses to the compiler to be used whenever a communications link to an assembly program or a dedicated location was required. The code generator then places the

global block or nameset at the specified location. Directives to the compiler take a simple form. For example,

```
/CHANNEL/ = 73 ;
```

puts block CHANNEL at location 73. The directives are on a file entirely separated from source code, source code of course has no access to the addresses, and the solution imposes no conventions or limitations on LITTLE source code.

The three preceding problems involve the world external to the code generator; its internal structure constituted the next to last major design problem. Though the top most structure of the translator could have been quite simple with just two passes over the parser output, one for resolving "forward references" and a second for code generation, the extravagant choice of four passes was made. Partly it was done to separate processes that were conceptually different so program development could proceed independently and so that debugging and compiler modification would be more straightforward. But it was also done to allow more chances for optimization; the longer one can postpone storage allocation, the better the job becomes. The philosophy here is simply that in implementation of a language for systems development, almost no effort by the compiler in behalf of optimization is too excessive. Not surprisingly, there was some expectation that the translator would run rather slowly. This did not, however, prove to be the case; the code generator accounts for about one-third of the execution time of the compiler. The functions performed in each of the four passes were:

1, symbol resolution to adapt LITTLE source into legal, non-redundant assembly language symbols,

2, a preparatory pass to determine the optimum code to generate and set directions for the succeeding pass,

3, code generation itself, and

4, a storage allocation pass.

This gross description of the structure is amplified in a detailed systems manual (3).

The final decision transformed the structure of the object output. The usual compiler converts a series of programs or routines or procedures into a series of independent object modules, to be linked by a loader. The translator creates a single assembly language program with multiple entry points. A single program causes all symbols to become global, so of course this creates work: the decision is the major cause of the first pass over the data, although some symbol resolution would have been necessary without it because, like any higher level language, LITTLE is not as restrictive in its use of symbols as an assembly language. The justification for the transformation lies, once again, in the optimizations it allows; these advantages are discussed later in the account of storage allocation.

Optimizing Machine Code

If the LITTLE goal of efficiency is viewed seriously in writing a code generator, then at least half of the result will depend on the care taken in choosing the machine instructions to implement each LITTLE primitive. For many machines a concurrent concern with register allocation is also important in reducing connective (loading and storing) code; for the accumulator machine here it is insignificant. Without going into any details of the code, the following paragraphs discuss a number of subjects that offer opportunities for optimization, and that might be of benefit in generating code for any language.

Varied Realizations

It is not uncommon practice to realize a primitive with a single macro or template, but for this LITTLE generator a single implementation is the exception. Most primitives give rise to several code sequences. In some cases the determinating influence is the code environment (preceding or succeeding primitives) while in others it is the characteristics of the arguments of the primitive itself. A good example of the latter type is the LITTLE field extraction operator. This primitive can lead to more than two dozen machine code sequences depending on the nature of the variable from which the field is extracted, the size of the field, its position, whether or not the size and position are constants, etc. Generally, for this primitive it is difficult to think of many situations where the assembly language programmer could turn out better code.

Space versus Time

Usually when two means to implement a primitive are available, one of them will be both shorter in code length and faster in executing. There will be no difficulty in choosing the better one. But occasionally

execution time and code space are in conflict. One way this may happen is when the code required by a primitive is a few instructions too long to be entirely reasonable in line, but as a run time library routine would require two or three times as long to execute. A typical solution is to allow the programmer to choose between the two possible optimizations at compile time; and the original design of this generator incorporated such a feature. Once the generators were begun, however, there appeared only three primitives that could pose any conflict of this sort, and two of them were rarely used. After hearing of a case where a version of a FORTRAN compiler that had been optimized to produce fast code actually turned out slower code than the unoptimized version, the design decision was reversed and the choice was made arbitrarily.

For library routines a substantial factor in execution time can sometimes be found in the transfer of arguments. Though an inefficient procedure may be chosen for other characteristics such as traceback facilities or inter-language communications, there is no reason to maintain the same procedure in designing library routines; several more efficient ones were used here.

Compile Time Optimization at Run Time

Every good compiler will take a division by two or a power of two and optimize with a register shift if the machine has no hardware for the operation. As a rule this optimization is confined to situations where the divisor is a compile time constant, but this need not be the case. If D is the divisor then the expression (7)

$$D \wedge (D-1)$$

is zero if D is a power of 2. It is, on the average, much faster to perform this test and carry out division only upon failure than to supply a somewhat shorter division routine lacking the test. Of course the same situation exists for multiplication, and an analogous optimization is possible with some of LITTLE's field operators.

Run Time Library

Several considerations here cause conflict in choosing a best implementation. On the one hand, the use of previously written machine independent code for the library routines is preferable because it requires no work. On the other hand, these routines are at the lowest level and an argument can be made for writing machine dependent, efficient code. An individual who has written the code generators has an excellent assessment of what source level code gives good machine level code, and what is impractical to express at the higher level. In this instance there might be about 100 words of assembly code in the run time library that could not be avoided, but the actual library has about 400 words on the basis of efficiency. Half the remaining LITTLE coded library was taken from previous libraries and the other half was re written mainly because of differing conventions.

Another aspect of the library mirrors the multiple realizations discussed above. Just as it makes sense to optimize with several code sequences in-line, so does it in the library. For example, both field extraction and field assignment operators can generate use of four different library routines.

Addressing

If one wishes to take advantage of procedures written in another language then adherence to the language's addressing conventions is helpful. In this case compatibility with FORTRAN was desirable, but its conventions were not. The point of dispute was the convention which stipulates that the address of an array is the address of the first element of the array. In every reference to an element, then, an extra operation is required to correct the address by one word. Moreover, the calculation can sometimes require a register and cause extra memory references for bumped temporaries. For the sake of efficiency it makes

good sense to define the address of an element (where the number of bits in the element is equal to the number of bits in a machine word) as either the sum or difference of the address of the array and the index to the element; whether memory is accessed up or down is not of consequence to efficiency.

Storage Allocation

For the Honeywell Series 16 machines storage allocation offers the compiler writer a real challenge. For computers that have homogenous memory and homogeneous addressing in the machine instructions, there will be no gains from any particular allocation scheme. For computers unlike the Honeywell but with addressing relative to the instruction, some of the problems and opportunities may be the same as those described here. Storage allocation does not often get serious consideration in compilers, but for a systems implementation language it should. Assembly language programmers only occassionally take the pains to optimize their storage, and for good reason: it takes too much time. But in writing a code generator for the general case the effort is made only once, and the excuse is insufficient. Herewith, some improvements.

Use of High Priority Memory

As noted earlier the target machine has a sector zero of 512 words that may be referenced from anywhere in core. With another computer the priority of a part of memory might be high because access is especially fast. In either case one is concerned with using the space to best advantage. Here, several disparate quantities were assigned to this sector. First were intersector references. Sometimes source code can generate significant numbers of cross sector addresses; even up to ten per cent of a sector can consist of these resolutions. Since there is much duplication among sectors, placing them in sector zero reduces code length. A second type of quantity is the literal. Some constants get repeated use, while others appear but once; it was a simple matter to keep track of the number of routines in which a constant occurs. If it appears more than once, it is assigned to sector zero. This

procedure would have been impractical without the global symbol, one program approach. A third category is the global variable. Though this is a natural choice, space in the sector is quite limited and might easily be exhausted, so only global variables no larger than the machine word size were assigned. In addition, some address constants pointing to word size arrays are placed here, again eliminating much duplication.

Juxtaposing Operators and Operands

This target machine is similar to a number of others in deriving space and time advantages from close proximity of operators and operands; specifically, if the operand is not in sector zero, there is advantage to having it in the same sector. Several tacks were made in this direction. The simplest was merely to allocate variables and temporaries to storage as soon as they were referenced in the code (actually after the first succeeding unconditional transfer). A second was generally to allocate storage for a nameset immediately after the routine in which it was declared. A third procedure created a new set of temporaries whenever a sector boundary was crossed. And the last procedure attempted to avoid loops over sector boundaries: on the average half the memory references in a loop will lie in the other sector. This attempt was quite difficult to implement and, although productive occasionally, was perhaps not worth the effort.

Minor optimizations

Two means causing slight improvements were an allocation check and a packing algorithm. The check merely prevented assignment of a variable to storage if it had not been referenced. Though newly written routines seldom contain enough instances to make it worthwhile, older, modified programs need some garbage collection. A simple packing algorithm proved

quite efficient in cramming small routines into sectors, almost always filling ninety to ninety five per cent of the space. Of course it is possible to emit code without regard for sector boundaries, but the number of inter-sector references rises so sharply that more rather than less space is required.

Unexpected Problems

There are always a few of these, but the list is not terribly long. The biggest problem was that the translator was rather difficult to debug. The most important causes were the difficulty of following directions given during one pass in a subsequent one, the lack of enough top down design in standardizing procedures or macros for code emission, and the slow growth of storage optimization features during the writing of the translator. This last cause gave rise to patchwork-like source and in one case a procedure got.so complicated it had to be abandoned and redesigned as a structured program.

A second problem was the parser. The translator was one of the first large application programs for the compiler, so a few bugs would not be entirely surprising. Except for one, they were minor. The exception resulted from a confusion over the word size in the parser host and the word size in the target code, and illustrated once again the difficulty in writing machine independent software. What was a surprise in the parser is that the only parameters needed by it at compile time to shift to another target were the new word and character sizes. One other change in the parser to produce more efficient code has been shelved; the loop dictions in LITTLE, such as do, while and until, are not primitives and the choice of primitives to implement them is not the most efficient on all machines.

Another source of problems was the assembler. Though this program had been executing for several years, it broke under the impact of LITTLE. Two principal causes could be discerned; one was the product of the extensive storage allocation mechanisms in the translator which employed many pseudo operations rarely encountered in the small programs

previously assembled; the other cause was the sheer size of LITTLE programs which overflowed the assembler's tables. During the difficulties it would have been possible (if not very convenient) to have jumped to the other assembler running on the target machine, but after some experience with the loader, there was no longer any confidence in official software.

Like the assembler the loader had been around for a while. Like the assembler it too cracked under the impact. To begin with it was a rather difficult piece of system software to use with complex input, but when it finally failed to meet its own specifications, a rewrite and extension was undertaken. The new code was written mostly in LITTLE, making this one of the few minicomputer loaders not in assembly code. The task was particularly painful in that, if a new loader had been foreseen, then it would have been written at the start, and the structure of the code generator would have been more standard with conventional global blocks and storage allocation.

The biggest surprise of all was the length of time needed to do it all - almost a man year.

Results and Expected Problems

Some test of the quality of code produced was desirable. There is, however, a difficulty in that any comparison of code among languages will reflect their biases. The choice of an algorithm should be grounded in some domain where the languages are at least not at their most awkward. The Heapsort algorithm of Williams (4) met this criterion for FORTRAN and LITTLE; it consists entirely of simple arithmetic, a few comparisons, and program flow statements. It was programmed in these two languages, and then the intermediate assembly code was inspected to find possible short cuts, giving an estimate of the shortest and fastest code obtainable.

Language	Execution Time	Code Length
FORTRAN	100	246
LITTLE	15	160
Assembler	13	138

FORTRAN is actually worse than shown in terms of code length, because the library routines it invoked were not included; LITTLE code was all in-line.

While the comparison between LITTLE and assembly code looks good, it is in reality even better. The test excluded two very important features that would be imperative to consider in an evaluation for systems programming. One of LITTLE's major assets is its bit and field operations. Even a good assembly language programmer will not always find the shortest code for each and every case of field manipulation. Second, and more important, no programmer has the time for all the global optimization undertaken and described in the preceding section on storage allocation. For a single, short routine such as the Heapsort test this does not come into play, but for large programs it is significant.

Taking account, there seems no reason to prevent the assertion: for systems programming the LITTLE object code may run faster than assembly code. It is certainly easier to debug.

Currently, efforts are nearing completion on implementing the extensive I/O of LITTLE and at least the most frequently used portions should be operational shortly. There are still no plans to implement real numbers. Conversely, the interrupt facilities of LITTLE are ninety per cent complete, and with some luck may become useful within weeks.

Two projects remain to complete the work on Series 16 LITTLE. First, the machine dependencies of the translator should be removed to allow execution on other hosts. These mainly result from poor character handling and could require two or three weeks to finish. Second, the generator should be extended to handle the PRIME computer, which includes the Series 16 machine language as an incomplete subset.

Brief guides to compiling for the Series 16 machines (5) and running on them (6) are available.

Acknowledgements

Day to day advice from E. Deak and D. Shields was a life saver, and the assistance of E. Guth with the code generators is much appreciated.

This work was supported by grant NS-10072 from the Public Health Service.

References

1. C. Weitzman, Minicomputer Systems, Prentice-Hall (1974). A discussion of software availability may be found herein.

2. Machine Oriented Higher Level Languages, W.L. van der Poel and L.A. Maarsen, eds. American Elsevier (1974).

3. T. Stuart, "The LITTLE Compiler for the Honeywell," internal report, Courant Institute of Mathematics, New York (1975).

4. J.W.J. Williams, CACM 7 (1964) p.347.

5. T. Stuart, "A Guide to LITTLE on the Honeywell Series 16 Machines," internal report, Courant Institute of Mathemtics, New York (1975).

6. T. Stuart "Loading and Running LITTLE on the H-316," internal report, Courant Institute of Mathematics, New York (1975).

7. D.H. Lehmer, "The Machine Tools of Combinatorics" in Applied Combinatorial Mathematics, E.F. Bechenbach,ed., John Wiley and Sons, New York (1964).