LITTLE Newsletter No. 21

December 4, 1972
David Shields

Some proposals for Improving Accessibility of LITTLE Compiler

In this newsletter we argue that sometime in the near future a "documentation interlude" is needed to substantially improve the accessibility of the LITTLE compiler to further change and improvement. We indicate how this interlude might be carried out; and also make a few miscellaneous comments about the documentation of the LITTLE compiler.

The "documentation " we describe in this newsletter is program documentation; that is, the source code for the compiler itself--the form and content of the comments and instructions in the compiler. We do not discuss the writing of manuals about the compiler, programmer's guides, etc: Such manuals are certainly desirable, but our intent here is to make the compiler proper as self-contained as possible.

By "accessibility" we mean the ease with which one can learn about and work with the compiler by examination of the source code alone; and also the ease with which modifications of parts of the compiler may be accomplished. Without affecting the entire compiler. Thus accessibility requires readability and modularity.

That the LITTLE compiler is not currently accessible in the sense described above is apparent from examination of virtually any part of it. The reasons for this inaccessibility are mostly historical, in that during the bootstrapping process, the compiler has gone through representations in at least three languages (LITTLE, COMPASS, and FORTRAN) over a time span of several years. Each such representation had added some "noise" to the compiler and has resulted in some loss of original algorithm of compiler.

The need for improved accessibility is apparent when we consider the demands to be placed on the LITTLE compiler once it has been successfully bootstrapped so that it is written in LITTLE itself. These demands include:

a) Generation of code for machines other than the 6600-IBM 360, Honeywell 516 (a minicomputer), and a "LITTLE" machine.

b) Improvement of basic block processing with addition of more machine-independent optimisation.

c) Possible implementation of LITTLE on other 6600 systems.

d) Refinement of current assembler part of compiler to produce better code for 6600.

e) Addittion of machine-independent optimizations based on analysis of program flow. Now the implementation of any of these projects would involve the modification or replacement of only part of the compiler--hence the need for modularity; and some understanding of the compiler so that the relevant subpart can be identified and isolated--hence the need for readability. Moreover, accessibility is necessary not only to make these projects less difficult; but to make them possible. For example, someone may be loathe to attempt to compile LITTLE for the 516 if it takes several man-weeks just to determine how the compiler works.

In the preceding paragraphs we have defined accessibility and shown the need for making the LITTLE compiler more accessible; we now discuss some techniques which might be used:

Improvements not affecting executability--comments,
variable and label renaming, lexical reordering
Improvements affecting executability--macro packing,
subroutine renaming, variable reordering,
machine generation of compiler subparts
Improvement of system interface
Debugging facilities
Improvement of transferability of compiler.

All of the improvements just mentioned are intended to improve accessibility without fundamentally changing the algorithms of the compiler. If any of these improvements are added, the new compiler obtained will be substantially the same. The improvements we propose are similar to those methods of transforming a program used for program optimisation. Optimisation transformations aim to produce equivalent programs which run in less time or use less memory; "accessibility" transformations produce an equivalent program which is more readable or more modular.

A critical problem is, of course, the verification that any change has not substantially altered the compiler, or added errors.

At worst we will have to run the new compiler against a (hopefully) comprehensive library of test programs; a test which requires a substantial amount of computer time; at best, as when we add comments, we need to verify that we have added only comments, and have not changed any executable instruction.

Consider a subprogram S which compilers to a binary module B. Consider S', which is obtained from S by any combination of the following transformations:

1) Adding or removing comments

2) Renaming variables

3) Lexically reordering the program; e.g., changing source so all statements start on column 11.

Let B' be the binary module obtained from S'. Then B' is identical to B, since none of the above transformations above have any effect on the binary module produced. Such transformations are best implemented on a subroutine by subroutine basis. They are verified by compiling the resulting routine, and comparing the binary module of the result with the binary result of the original source on a bit-by bit basis; accepting the change if no bits differ.

Lexical reordering is particularly important when a program has been modified by several people over a long time span: Lexical reordering is best done by a program, which reads a program as input and produces an equivalent program in which statements and expressions are arranged in a standard format. An example of such a program is TIDY, available at NYU, which reformats FORTRAN programs. Such a reordering is attempted in a minimal fashion in producing the "punchout" file of the LITTLE lexical scanner, in that

a) statements begin in column 11
b) labels are started in column 1 (when label defined)

Perhaps the greatest problem in constructing a reordering program for LITTLE is how macros are to be handled. In general we would want to keep macros unexpanded; however, renaming of lables and variable is best accomplished by using macros. Thus it seems desirable to add an option to the compiler in which only selected macros are to be expanded. Also, if some macros are not to be expanded, then we cannot attempt to parse the program to recognize statement boundaries, subroutine definitions, label definitions, etc. For example, if the program uses the macro

```
+ * LABCHK (L,TEXT) = /L / PRINT TEXT; **
```

then the reformatting rules (a) and (b) above will not suffice to isolate statements and label definitions. The best approache seems to use standard rules in formatting macro calls, so that macros of the form above need not be processed.

Transformations which produce a different binary module must undergo a more extensive verification procedure, the new module must be executed and the resultant output verified. To minimize the time necessary to carry out such tests, it seems advisable to distinguish changes which involve only a single module or small group of modules from those which potentially may alter the entire compiler. Examples of the simple changes, which may be verified by running the new module against a small test library, are:

a) renaming of subroutines and functions
b) reordering of variables,i.e. changing order in which variables SIZED.
c) Very local changes to a routine, e.g. changing the content of an error message

Examples of changes which require more extensive verification are:

a) Consolidation or redefinition of several modules to improve modularity.
b) Change and or clarification of system interface.
c) Change in any fundamental compiler data object, i.e. adding new field to HA or VDA.

Such changes require the possible recompilation of the entire compiler, and verification against a large test library.

A further sort of transformation, which may fall into any of the above classes, and which seems particularly important, is the "recognition" of macros; that is, the recognition of code patterns which can be realised by expansion of macros. The resulting consolidation of code obtained by defining new macros whenever possible, improves readibility in an obvious way, and aids modification of the compiler.

Another means of consolidating code is to use programs to produce repetitive or highly structured parts of the compiler: Two candidates for this method are the parser proper, and the error-massage routine.

The LITTLE parser (routine SYNREC) was originally produced by a FORTRAN program which accepts a description of th grammar in a Backus-like form, and produces a FORTRAN parser. Historically, changes to the LITTLE grammar, have been effected by changing the parser itself, so the original (and much more comprehensible) Backus grammar has been lost. Now, only slight changes to the meta-compiler (which is available) are needed to produce a LITTLE parser(due to availabilty of macro-processor). The use of the meta-compiler would ease changes to LITTLE grammar, and would provide for a much sounder definition of LITTLE syntax. Similarly, the error routines invoked by the parser could profitably utilise a message-table generator. This would facilitate the change of message text, and the addition of new messages when the grammar is changed. Such a generator program is very simple, and is used in the SETLB system; the SETLB generator program accepts input data of the form 1 "error 1--bad statement, missing semiclolon" and produces (FORTRAN) DATA statements for defining a message-array, e.g.

```
DATA (INDEX(1) = 10)
DATA (TAB(10) = 10HERROR 1--,)
...
DATA (TAB(14) = 10HSEMICOLON)
```

An important area of improvement is the identification and isolation of parts of the compiler which interface with the current operating system. For example, the compiler contains the statement

```
CALL FINBIN (1,0)
```

which results in an efficient packing of a binary token file used for communication between lexical scanner and parser. Statements of this sort should be clearly indicated and isolated; otherwise substantial problems in portability will result.

Further areas of relevance to portability issues are the following:

a) Isolation and indication of memory management; so that size of a compiler array may be easily changed: This requires knowledge of variables defining size and dimension of an array, and, if array used to contain indices, or pointers, the size and definition of pointer-accessing parts of the compiler.

b) Some provision for maintaining the source in a Library form; this is perhaps best achieved by constructing in LITTLE a library maintenance routine similar to the CDC program UPDATE.

c) Provision for isolating and handling character-set problems; for example, how to hang the semi-colon on other machines, how to convert character strings.

d) Definition of operating system interface needed for implementation at other sites, both those with 6600's and those with other hardware.

Also, the debugging facilities within the compiler should be strengthened and extended, if possible. For example; tracing of loads or stores of key fields in the HA and VDA should be possible (this is possible in the current FORTRAN bootstrap, but not in the LITTLE written version).

In summary, we have indicated the need for making the LITTLE compiler more accessible, and have indicated some approaches to use and the gains to be expected. These changes are best accomplished by a "documentation interlude" in which only changes of this sort are performed. Once the compiler has been clean up, modifications and improvements should proceed at a much faster rate than would be otherwise possible.

Appendix-to LITTLE 21

Comments by
Kent Curtis

1. On macros:

macros have three uses in LITTLE.

a) operator definitions, e.g. FIVEARGS
b) parameter definitions
c) equivalence declarations

In cleaning up the code type C macros should be eleminated and a mininuim set of variable names used consistertly throughout all parts of the code.

Type (b) macros should be isolated in a single bloc at the beginning of the code with a comment on each explaining the nature of use of the parameter

Type (a) macros should be isolated in a single bloc at the beginning of the code, some of the existing ones are more confusing than useful, e.g.

+ * GOBK = GO TO SYSBACK **

and should be eliminated. No macro definition should be embedded in the code.

A fairly simple program should be able to read the compiler code, make desired substitutions, and write revised code in standard format.

2. On code organization.

code should be reorganized to show:

1. Explanatory comment on overall compiler organization and operation, defining parameters, principle sections, and linkages.
2. Macro block refining parameters
3. Macro block defining operators
4. Initialization blocks (well annotated)
5. Main program
6. Principle sections in execution order

7. Subroutine or function definitions in alphabetic order.
8. Interface to host system e.g. CALL FTNBIN should be replaced in code body by call to something defined here.

machine dependency should then be isolated to items 2,4 and 8. above (and explained in item 1). One might hope, in fact, to isolate machine dependence to items 2 and 8.

3. Proposed modus operandi

a) Finish bootstrapping compiler as is.
b) Then clean up code a la suggestions
c) Write LITTLE machine simulator (LMS) in LITTLE
d) Write machine dependent LMS → host machine translator
e) Bootstrap entire package on 6600
f) Bootstrap entire package on 360/ ? or 370/ ?
   Revise as experience dictates to complete isolation and explanation of machine dependency
g) Do comprehensive test on both machines including making compiler modifications to analyze ease and indiot-proofness. Revise as experience dictates.
h) Prepare a systems programmers reference manual and a users manual
i) Give it to the world and elaborate the compiler itself as desired.