LITTLE Newsletter # 42

T. Stuart
July 20, 1976

A PARSER-CODE GENERATOR INTERFACE

With more code generators for LITTLE being planned it is appropriate to re-consider the data structures linking the parser and the generators, especially the infamous VOA. The complexity of these structures (six or seven tables and two or three score fields) requires substantial time for familiarization for programmers who wish to write new LITTLE code generators. The use of the generators by parsers of other languages is also inhibited.

A series of independent steps or projects is suggested that will lead to easier access. Replacement of the VOA itself is no longer feasible because a parser and four generators rely on it. Rather, the approach is to hide the data structures entirely from users. At a later stage alternative, more efficient structures can be chosen. The tasks proposed are:

1. Create a module which reads the tables.
2. Create another one which writes them.
3. Design efficient, compact structures.
4. Map the interior structures to an exterior representation - an intermediate language.
5. Create a symbol customizing module for generators that produce assembly code.

The first of these is the only topic of this newsletter. The others are dependent on the first and will be treated in subsequent newsletters. For those readers with only a cursory interest in the problem, it is suggested that you skip immediately to the results. Pages 23-26, of the Appendix, Newsletter 42A, give examples of how new code generators can be written.

<u>Overview of the Code Reading Module</u>

If the structures are to be hidden from code generators, then two separate courses may be taken in providing information about the operators and operands. The first is a functional approach in which a large set of access functions may be designed; each function returns some property when invoked. The second is a windowing approach where a command from the generator causes display of many properties in a window, or set of global variables. Choosing a window size (i.e., the number of properties) can be a problem. If there are too many items one is hardly better off than with the present complex structure, and if the kinds of data to be windowed are enormously varied then the number of windows needed may also be too large for clarity.

A functional solution, on the other hand, poses several design problems. Typically code generators must access a three level structure: a property of an argument of an operator. Also quite typically, they will often need to reference two such items at once; for example:

```
if size of argument 1 of operator 1 =
   size of argument 2 of operator 2 then ...
```

There are three means of allowing this double access within a single expression.

First, once can introduce functions designed explicitly to set the current operators and arguments, while yet other functions access specific properties. Not only is this quite cumbersome in practice, but it will be easy to introduce unpredictable side effects. A second method would allow the access functions to read global variables in retrieving a property. This method lends itself to simplicity and brevity in the assembler code, e.g., if fsize1 ( ) = fsize2 ( ) then..., but suffers the defect that multiple sets of functions are needed to access multiple sets of global variables. The third means is to design functions that neither read nor write global variables, but have all such information specified in the calling parameters:

```
if fsize1(op1,arg1) = fsize2(op2,arg2) then...
```

Obviously, brevity of code will not be an asset of this approach.

Overhead is another factor to consider in choosing between a functional and a windowing solution. If the number of different properties referenced with each encounter of an operand is small, then there is a lot of overhead in displaying all the others, whereas if the number of properties referenced is large, there are many functions to invoke versus only one window. I think this is the crucial factor: on one hand the overhead of the window approach is all in extra execution time with little programming for the writer of the generator, and on the other hand it is all in the programming if the functional approach is taken. Our experience with four-pass generators suggests that multiple pass costs are trivial.

What Questions Do Code Generators Ask?

The kinds of access needed by generators might be divided into six categories

1. Sequential access to operands.

In this case there is no concern for operators. Examples might be a routine that converts LITTLE symbols to legal assembler symbols or a procedure for storage allocation.

2. Access to a single operand of an operator.

This type of access takes two forms, specific and sequential. Specific access is by far the more frequent request; most work in code generators deals with operands in a manner where the purpose or function of the operand is germane, and access to it must reasonably take into account whether it is an addend, a field width, a label, etc.

Sequential access to each argument is often most appropriate when the semantic class or purpose of the operand is of no concern; when only a property or attribute is important. For instance if a check is made on a relational operator for multi-word arguments in a pre-emission pass, the left and right operands are of no interest as distinguishable quantities, and sequential access to the operands of a particular operator is sufficient.

3. Access to all operands of an operator.

This is clearly required for code emission proper. A simple one-pass template generator may be able to get by with this single facility.

4. Comparison of operands in different operations.

This activity is most frequent in trying to optimize code where the output of one operator is the imput to another. Thus, the second operand is usually found by a sequential search over the operator space following the first primitive.

5. Random access to operands

Several instanceswill occur during code generation where operands of a given type or group will be treated collectively. For example, a code generator might declare all external symbols in a contiguous list. If the generator is to avoid perusing the entire stack of operands every time it treats collections, an interface must provide some facility for random access.

6. Access to all operands of two operators.

This situation arises often enough to be of concern. Most common is the construction

```
IF A > B THEN ...
```

where better code will be obtained on many machines if the occurence is dealt with as a joint operation rather than two separate ones. However, the general solution for allowing access to all operands of both operators implies too complex and too rarely invoked a procedure to justify inclusion in a general module. We leave this situation untouched, though it should be clear how a code generator could include the extension.

Operator Descriptions

The most important thing about an operator is clearly its type, and we have stolen 74 macros used in the LITTLE parser. A few new operators have also been added: the subr entry, the fnct entry, the begin, and the end operator. Their inclusion will simplify code generation. Space is reserved for two other primitives, a block definition function to be fleshed out later by the optimizer, and a comment operator to tie source code into assembler code for aid in debugging.

The number of arguments of an operator is a second important characteristic; in fact it is crucial in the case of operators with a variable number of arguments (subr entry, fnct entry, subr call, fnct call, data assignment and goby). Referring back to the second situation (one by one operand access) however, it seems advantageous to provide an argument count for all operators (rather than just those that require it) to aid in sequential operand retrieval.

A third item needed in generation is the position of an operator in the code buffer, but when this is important it seems so only in the sense of being adjacent to or close to other operators. No case appears where access to operators in a random manner is required. (Though the current generators do frequently access the VOA in a random fashion, these instances are operand references.) Thus, this fact suggests that a stack mechanism is the appropriate one for operator access to the code buffer; no operator pointers are needed.

The only other operator information we need, strictly speaking, is the characteristics of its operands. These will be windowed by display requests and are described in the next section.

The questions asked by code generators will require two stacks and the second must be invoked relative to the first (for searches over the adjacent operator space.) The following commands, implemented as macros, can take care of these needs.

code-init;

The purpose of this command is to initialize both operator stacks to point to the first primitive, the *begin* operator. An argument stack, described later, is also initialized.

pop;

This command pops the next operator off the principal code stack, setting two global variables, *op* and *count*. *Op* is the operation type, and *count* is the number of arguments. No corresponding push command here seems useful. The command also initializes a hidden pointer to a sequential list of the operator's arguments; sequential retrieval may then be performed with the *get-next* command. Yet another initialization is of the hidden pointer for subsequent search of the operator space adjacent to *op*. This pointer is used by the succeeding *pop 2* and *push 2* commands.

pop 2;

This command pops the subsidiary operator stack and sets the global variables *op2* and *count2*. The starting position of this second stack is re-initialized with every *pop* executed.

push 2;

This searches the operator space in the reverse direction of *pop 2* and also sets *op 2* and *count 2*. A previous *pop 2* has no effect on *push 2;* both commands function relative to the current *op*. They are therefore not identical to the usual type of push or pop stack commands. To define them in a standard way introduces some minor complications that are not justified by the simple needs of code generation.

These four commands and four global variables constitute a complete account of generator access to operators. Operands will not be so easy.

Displaying Operand Properties

The type of an operand is often the first question a code generator must ask. According to the different actions a generator must take, eleven semantic quantities must be distinguishable by a code generator. We list them with the internal representation of the corresponding value of the operand.

Quantity	Representation When Target is Host	Representation When Target is Not Host
Bit String	Bit String	Bit String
Negative Integer	Bit String	Host Format Number
Real Number	Real Number	Character String (C.S.)
C.S.	C.S.	C.S.
Character String Code	C.S.	C.S.
Temporary	No value	No value
Variable	C.S.	C.S.
Formal Parameter	C.S.	C.S.
Label	C.S.	C.S.
External	C.S.	C.S.

Note that the above values are not those presently in use. In addition, to the quantity type and value of an operand, the size, the number of bytes or characters, the dimension and the nameset are static properties applying to some of these types and an operator - bound property is the use of the operand. The use (more precisely, the scope of use) of operands in LITTLE is now defined only for temporaries and no immediate extension is required; if the temporary is used as an input argument to an operator its use is 'true' if this is the final use. As an output argument, the temprary's use is the number of subsequent operations between the output and the final use. Note that this is a slight change from the *lastuse* field of the VOA.

Unlike operators, random access to operands is sometimes necessary, If the code generator creates its own data structures with partial or additional operand information, random access will be frequent. To accomodate the need, a unique *id* is displayed whenever an operand is windowed. For practical purposes it is not significant whether the current HA location or the current VOA location or any other convention is chosen.

To summarize the argument attributes are: the value, the quantity type, the id, the size, the dimension, the nameset, the number of bytes, and the use.

The Classes of Operands

Operands are classified according to their semantic function in an operation. The intent is to allow systematic treatment of operands and encourage more easily understood source code by imposing a class. With a couple of exceptions, pointed out below, the imposition is neither unnaturally coercive nor ambiguous. The definition binding operands to classes is given in full in a LEGAL table in the appendix, newsletter 42A. Here we summarize the classes.

Source Class:

For each operator where there is a single principal input operand it is assigned to the source class. This includes unary operators, all assignments,extractions, ifs, and unformatted output. For unformatted input, the external file is the source.

Result Class:

This is always a temporary. Every operation which creates a temporary will have one result operand.

Target Class:

This is usually a local or global variable. It occurs in all assignments (including data assignments) and unformatted input. On unformatted output, the external file is the target.

Index Class:

For every indexed reference the operand will be in this class; so too is the index of data operations and the indexed transfer (goby).

Left Class:

In all binary operations this is the left operand. In unformatted I/O this is the left (lowest) index of a dimensioned variable; there is some slight forcing of the definition here.

Right Class:

The right operand in binary operations and the higher index in unformatted I/O belong to this class.

Location Class:

An operand in this class is either a label in the goto, label, and if operations or it is a routine name in a call or entry operation.

Width Class:

The number of bits in field operators and the number of characters in self-defined string operators are in this class.

Origin Class:

The corresponding start-of-field postions for the same primitives found in the width class are assigned to this class.

Multiple Class:

This applies to all subroutine and function entry parameters, to all subroutine and function call arguments, to all constants in data operations, and to the labels of the goby primitive. If the designer of a code generator finds the treatment of this potpourri in an identical manner obnoxious, then macros can be defined to simulate more classes.

Windowing an Operand

The display of the eight argument attributes could be done in three ways. Each attribute could be assigned to a simple variable, to a field in a multi-word variable, or to an element of an array.

The first has the disadvantage that, since the number of windows proposed is not small, the number of variable names will be too

cumbersome. There is no overriding reason for choosing between one or the other of the latter two data structures. Field use in multi-word variables is somewhat more error-prone, while array use suffers from the fixed size of its elements. Since the only one of the attributes that can be unduly large is the value of an operand, and since it can vary from 1 bit up to 2000, provision of space for the value itself in every window is too expensive. (In addition, the extraction of a character string from a field is a relatively awkward activity.) So long as values are not always to be displayed, then the array representation for windows seems slightly preferable. The size of window arrays is implementation dependent, but would normally be the host machine word size. We choose the convention that when a value is too large to display in the *value* element of the window, access may be had through a *valu* subroutine, the arguments of which are the operand *id* and the returned value, *V*: call valu (id, v)
The subroutine will return the value, *v* whether or not it also appears in the window. When a bit string value is returned it is the responsibility of the writer of the code generator to insure that *v* is sized large enough to accept the value; when the value is represented as a character string the generator must pre-set the origin field, and subroutine *valu* will test it magnitude, reporting an error if it is insufficent to hold the argument. In general, then, the only quantity types that will frequently be directly accessed though the *value* element are bit strings, negative integers, and (when host and target machines are identical) real numbers.

One other function will be convenient to supplement the windows. The representation of the nameset element, called *nsid*, has not been specified. Though a self-defined string would be quite adequate, the size of window elements on some host machines may be too small to be useful here, so we always stipulate that *nsid* is an identifier which produces the character string when used as an argument of a subroutine *nm*

```
call nm (nsid, nameset-name)
```

<u>Window Generation</u>

To decide how to formulate the opening of a window we look back to the questions generators posed. For each situation a command to supply the answer is specified here.

1. Sequential access to all operands.

Operands may be addressed in succesion by popping a stack. Initialization of the stack is performed with the aforementioned command -

code - init;

Popping the stack or que is done with -

pop-queue;

The window holding the operand is called *que*. The order of operands is undefined, though every operand will appear once. The operator bound attribute (the *use*) is undefined. When the stack is exhausted, all attributes of an operand are once again zeroed. As an instance of the use of this command, consider a search through all operands to find those with a size greater than the machine word size. Example 1 shows such a search and may be found in the appendix, newsletter 42A , to the present newsletter.

2. Access to a single operand of an operator.

The window opening command to retrieve an operand of the current *op* is -

get-arg (o);
or
get-arg (class);

The argument of the command is either the semantic class of the operand or zero. If a class is specified, then the window named for the class is opened. Should the specified class be an undefined one for the operator currently displayed on the stack, then the window is zeroed. Successive commands with the same class are redundant except for the *mutiple* class. With this one each command produces the succeeding operand on the list; when the list is exhausted all attributes are zeroed. The *multiple* queue or stack is initialized when the current operator is first produced with *pop*; no means of re-initializing is provided. Example 2A shows how a code generator determines whether the result of the current operation is used only in the next one.

If the argument of the *get-arg* command is zero, then the next operand in the current operator's list is shown in a window named *arg*. The operand queue is initialized by the *pop* command and, as above, no re-initialization is possible. The attributes are zeroed after the exhaustion of the argument count. The *arg* window and the class windows are independently generated; thus the same argument could appear in two windows at once. An *arg* window could be used effectively to check all operands for a characteristic. Example 2B shows a test of all operands of an operator for the presence of formal parameters.

3. Access to All Operands of an Operator.

This command opens each of the windows defined for the current *op;* it affects no other windows. The command is -

```
get-all;
```

In the case of operations with operands in the *multiple* class, only the first one is immediately available. Example 3 shows a typical framework for the principal code emission loop.

4. Comparison of Operands in Different Operations.

This command relies upon previous definition of the second operator, *op2*, with either the *pop2* or *push2* commands. The access of the second operand is accomplished with

```
get-arg2  (0):
get-arg2  (class);
```

and it places an operand in the *arg2* window. The command argument, like the *get-arg* command, may be a zero or a operand class. Unlike the earlier command, however, the corresponding class windows are not defined; only *arg2* is available. This command is a no-op if a *pop2* or *push2* has not been executed after the last *pop*. It zeroes the window if the corresponding operand class is undefined for the current *op2*, and it also zeroes all attributes when a *multiple* class or a command argument of zero has exhausted the queue. This command finds use in code generation when two LITTLE primitives are combined or when subsequent operations are searched for a use of the result of an earlier one. Many machines have "jump on condition" operators, and example 4A shows a search for a conditional transfer primitive following a *op-gt* primitive. Example 4B is a routine that searches subsequent primitives and counts the number of uses of a result temporary.

5. Random Access to Operands.

Any operand can be windowed by invoking the *get-any* command with a previously saved *id* as argument -

get-any(id);

The operand is displayed in a window called *any*. Like the earlier *que* window, operands here are not operator bound and hence do not have their *use* attribute defined. An example here seems unnecessary.

Code Generator Control

A LITTLE code generator receives its input from a file containing a sequence of routines. Opening this file, reading routines into the data structures, and signalling the end of the file is all carried out by a function called *another*. This routine takes a dummy argument and returns a true value upon finding another routine, and a false value otherwise. It also performs an implicit *init-code* command. Example 5 shows a possible uppermost level routine of a 3-pass generator.

Summary of Commands

Command:	Function:
init-code	initializes stacks for *pop* and *pop-queue.*
pop	pops next operator into *op* and *count.* initializes stacks for *pop2, push2,* and *get-arg.*
pop-queue	windows next operand into *que.*
pop2	pops operators in stack following *op.* initializes stack for *get-arg2.*
push2	pops operators in stack preceding *op.* initializes stack for *get-arg2.*
get-arg()	windows an argument of the current *op.*
get-all	windows all arguments of the current *op.*
get-arg2()	windows an argument of the current *op2* in *arg2.*
get-any(id)	windows operand *id* in *any.*

Summary of Windows

Operator-bound and classified windows:

source	*result*	*target*	*left*	*right*
index	*location*	*width*	*length*	*multiple*

Operator-bound but unclassified:

arg *arg2*

Independent windows:

que *any*

Summary of Windows for Each Operator

begin and end

none

fnct entry

location, multiple (number of multiples=count-1) subr entry

location, multiple (count-1)

addition (same for all binary operations)

left, right, result

not (same for all unary operations)

source, result

fnct call

location, result, multiple (count-2)

subr call

location, multiple (count-1)

simple assignment

source, target

data assignment

target, index, multiple (count-2)

field assignments

source, target, width, origin

binary or unformatted I/O

source, target, left, right

return

none

field extractions

source, result, width, origin

if

source, location

goto

location

goby

index, multiple (count-1)

indexed load

source, index, result

indexed store

source, target, index

indexed field assignments

source, target, index, width, origin

Structure of the Interface

Some of the interface structure has been implied by the previous discussion. Apparent characteristics governing the choice of interface design are:

1. Ease of Coding

This is the paramount objective, and the examples of the appendix should suffice as a measure of the success of that objective.

2. Hiding the data structures

This allows alteration of structures without impacting code generation and new data structures will in turn lead to more efficient use of memory and adaptability to dynamic storage allocation. The hiding of data also leads to a prohibition on addition of information to the parser created structures. Though all present code generators now do this routinely, I view it as their principal defect. Forcing the code generator designer to choose their own data structures can only make them more appropriate for their purpose.

Several other influences or aspects of the design may not be quite so obvious as the foregoing two. First, the interface itself is layered; all macro, function, or subroutine calls in a code generator involveupper level routines which perform error checking on the calls and parcel out the work to lower level routines. The upper level is also oblivious to much of the detail of the data which it aids in retrieving: this will facilitate addition of new primitives, operator classes, quantity types, or argument attributes with a minimum of effort. (By a minimum I mean only adding a macro or two; in complex cases, like converting a LITTLE interface to a FORTRAN interface, it might mean a day's work.)

Implementing the Interface

Since the upper level parcels out retrieval work into a large number of very small tasks to be performed by the lower layer, it means there are many lower level routines to be written for any given data structure, 26 in fact. Only one of the 26 is more than a few lines, and the group as a whole could probably be written and debugged with a week of work, assuming an understanding of the data structure and of the interface. The specifications for all the routines are given on pages 20-22 of the appendix. The one non-trivial routine is rd-routine. Though most of its work is simply reading a routine from a file (and for the VOA such a routine already exists) this program might also map current VOA bits and fields into those required by the definitions in the interface. One mapping seems highly advisable: conversion of the *lastuse* field as presently defined to the new one. Other mappings are more optional, and a determination should be based on whether the pre-processing significantly reduces the source code and execution time of the rest of the lower level routines. It should be emphasized that the one week estimate above is valid only for someone familiar with present data structures.

Comments

Richard Kenner is due thanks for reading and commenting on this newsletter. The possible drawbacks to the interface, he suggested, were an increase in code length and an increase in execution time.

The length of the interface routines will be approximately 1500 words of 6600 memory; moreover this increase would be part of a permanent root for any generator constructed with overlays. This length is not horrendous, but any additional memory requirements must be viewed with dissatisfaction because length is one of the LITTLE compiler's main problems. However, the apparent increase will probably be illusory; a decrease is expected for several reasons.

First, the extraction of information from the data structures is now a cumbersome, inefficient, and repetitive process. For example, the size of a quantity is obtained with this expression:

syze vod(q)

This is a field extraction from a multi-word indexed array. Inspection of the code generators from the CDC and the Honeywell machines showed 80 and 95 occurences respectively. On the other hand in the proposed interface, the extraction of this datum occurs but once, and all subsequent references require only an indexed load from a word-sized array (and the index is even a constant). For most computers the present method will lead to code at least three times longer, and may even involve a call to a library routine. Many quantity attributes are even more complicated to access than the size, though few are accessed more often. An accurate estimate of the decrease in code length caused by adoption of the interface would be difficult. Nevertheless, it would not be surprising if this single reason - the simplicity of attribute access - more than compensated for the entire 1500 word interface.

A decrease in code length can also be expected if and when the data structure is replaced. The design of a new structure is outside the scope of this newsletter, but it is reasonable to expect that it will be much smaller, and will be generated by a program that copies parser-voa output to the new structure before the code generator begins execution. I estimate that a well designed structure could be 25% of the present length, a saving of about 8000 words. Redesign of the data structure to effect a reduction in its size is, of course, possible without the superstructure of the proposed interface. However, without hiding the structure - one purpose of the interface - it becomes rigid as soon as two generators use it. In addition, an interface allows one to design a more complex structure than the constraints of multiple use by many programmers would allow.

A third reason for code length reduction may lie in the introduction of dynamic storage allocation to LITTLE. It is impossible for me to see a practical dynamic implementation of the current data structure, while the new one should provide for it.

Moving on to the other drawback, execution time can be expected to rise for a couple of reasons. The command structure is designed to produce all attributes of a quantity whenever any information at all is retrieved, obviously an inefficient action if only one attribute is needed. Secondly, the interface will be inefficient because it adds three or four subroutine levels to the generator; it was pointed out that this will be particularly inefficient on an IBM 370.

Taking the overhead in the command structure first, it is questionable whether serious overhead does exist. A one-pass generator, for example, needs almost all the attributes in any case. Even when multiple passes are made, most of the attributes still seem to be needed - at least in the case of the Honeywell generator. Also pertinent here are repetitive accesses to the same attribute as found in present code generators. Repetition is encouraged by the nature of the code generator programming process: an initial, crude generator is progressively refined to yield a better product, and the refinements lean towards additions (with repetitive code) rather than re-programming. Though some of this repetition may be removed from executing code by the optimizer, the inherent inefficiency for the programmer of repeated expressions can not be affected at all.

The interface, it is suggested, will also slow down a generator because of the number of subroutines introduced. This decrease can only be charged to the calling process, i.e., can only be expected on machines that impose a penalty on calls (by saving all registers). Otherwise, as has been argued earlier, the increase in code in the interface will be offset by reductions elsewhere. The interface itself has few loops; they are short; and the only two inner loops are the shortest. It cannot be denied that execution time will suffer on machines that penalize subroutine calls, so it is worth re-examining the command structure to try removing some subroutine levels.

Since the interface is addressed with command macros that call upper level subroutines, the calls could be replaced by the text of the routines. This route is unimportant when the command is rarely invoked and is not practical when the subroutine text is long. These two considerations eliminate commands *init-code, get-arg, get-arg2,* and the two upper level, directly called routines *valu* and *nm*. The remaining command macros are *pop, pop2, push2, pop-queue, get-all,* and *get-any*. The replacement code in all these cases is between 10 and 15 instructions. The only significant disadvantage in carrying out these macro expansions is to open the interface to intrusion from the code generator (since all variables used by the interface are now either in private namesets or are subjected to error checks when provided by the code generator). The lessening of this protection seems outweighed by the efficiency advantages of the macro redefinition. Richard Kenner is thanked for this suggestion.

On the other side of the interface, at the lowest level, many of the 26 routines that address the data structure are called from only one location in the interface, suggesting that simple code expansion in the calling routine would be practical. However, the first implementation will be done with the present VOA structure, and the current layered design allows an easy transition to a second implementation based on a new structure. Once this is done and if efficiency is still a concern, then the layer can be removed.

The question of speed for code generators must be put in perspective. The two faster generators now account for 20 and 30 per cent of LITTLE compile time. Both had their speeds pumped up substantially after spy tests located poor I/O practices and excessive use of multi-word routines. The other generators can be expected to improve when similar studies are made. Hence, our situation has some padding: every 10 per cent increase in generator run-time leads to only 2 or 3 percent increase in compile time.

There is one final point that should have been made in the opening paragraphs of this newsletter. While the present generators are layered on the output side (the code emission side) they access input directly from the upper layer of subroutines. Another way of saying this is that while output is relatively well-structured, input is not.