LITTLE Newsletter # 28

S. Brown
October 12, 1973

An Intermediate Language for the LITTLE Compiler.

This newsletter is intended to describe an intermediate language which will be used as input to the LITTLE compiler. For certain applications such as generating LITTLE from the BALM compiler it is desirable to bypass the LITTLE lexical scanner and parser. Presumably correct LITTLE is generated and a syntax check is unnecessary. Ideally an intermediate language should be general enough so that it is not affected by internal changes to the compiler. However, it should also be specific enough so that it can easily be processed.

For the LITTLE compiler output from the parser is in the form of entries into internal tables used by the assembler to generate machine language. The parser calls generator routines to make entries into the tables. It communicates with the generator routines via a stack. The intermediate language is made up of directives to place variables and constants on the stack and action directives which generally correspond to generator routine calls.

The intermediate language , hereinafter refered to as LIL, (LITTLE Intermediate Language) will consist of a stream of tokens which are interpreted as action directives and arguments. The following table presents a list of these actions.
This table is presented in two parts. Part A consists of directives to make entries in a symbol table which is more or less the counterpart of the LITTLE table HA.

Part A

ACTION	ARGUMENT 1	ARGUMENT 2
1. Add Name to symbol table	# of Characters	Value
2. Add String to symbol table	# of bits	Value
3. Add integer to symbol table	# of bits	Value

Part B consists of actions which reflect LITTLE language capabilities. The arguments for these actions reference entries in the table constructed by the Part A directives.

Part B

ACTION	ARGUMENT 1	ARGUMENT 2
1. Push onto stack	Symbol table reference	
2. Subroutine	Symbol table reference	
3. Function	Symbol table reference	
4. Label	Symbol table reference	
5. GØ TØ	Symbol table reference	
6. GØ BY	# of Labels	
7. Argument	Symbol table reference	
8. IF	Symbol table reference	
9. Call	# of Args	
10. Return	------	
11. Size	Symbol table reference	integer
12. Dimn	Symbol table reference	integer
13. Assignment	Type 1. simple 2. Indexed 3. Field extract 4. Field extract Indexed	
14. Binary operator	Type 1. + 2. - 3. .GT. 4. .LT. 5. .GE. 6. .LE. 7. .Eq. 8. .NE. 9. .* 10. / 11. .OR. 12. .C. 13. .AND. 14. .EX.	

ACTION	ARGUMENT 1	ARGUMENT 2	ARGUMENT 3
15. Function call	# of Arguments		
16. Monadic operator	Type 1. .NB. 2. .FB. 3. .Not.		
17. Field extract			
18. READB	# of elements in IO list		
19. Read	# of elements in list		
20. WriteB	# of elements in list		
21. Write	# of elements in list		
22. Endfile	integer		
23. Rewind	integer		

Directives from part A and part B may be intermixed in a file of LIL as long as the name or constant is added to the table before it is referenced.

LIL is essentially parsed LITTLE. In some cases the actual operand for a LITTLE expression is included as an argument following the action. For example:

```
GØ   TØ  LBL;
```

is expressed as

```
GØTØ     LBL
```

In other cases operands are pushed onto the stack before the action. For example:

```
If (X .EQ.  Y)  GØ  TØ  LI;
```

is expressed as

```
Name            1         X
Name            1         Y
Name            2         L1
PUSH            1                         (X)
PUSH            2                         (Y)
BINARY op .EQ.
IF              3                         (L1)
```

The only exception is for SUBR and FNCT definitions where the arguments are specified in LIL following the definition action. For example:

```
SUBR       ABC(X,Y);
```

is expressed as

```
Name            3          ABC
SUBR            1                          (ABC)
Name            1          X
Argument        2                          (X)
Name            1          Y
Argument        3                          (Y)
```

It is assumed that the generator routines will pop the stack appropriately, replacing their arguments with their result. Thus pop instructions are not part of LIL. For example:

```
A = B * C + 1;
```

is expressed in LIL as follows:

```
Name          1          A
Name          1          B
Name          1          C
Integer       60         00000....01
Push          1                    (A)
Push          2                    (B)
Push          3                    (C)
Binary op     *
Push          4                    (1)
Binary op     +
Assignment    Simple
```

The most convenient representation of LIL is as a binary file where each action is an integer code taking a field word. A coded file of fixed field format would be easy to handle also. Since the particular representation requires changing only the WRITE routine in the program which produces LIL and the READ routine which will be added to the LITTLE compiler, its exact form may be determined later.