March 1982

CIS-TR-82-01

A SESSION WITH THE LITTLE META TRANSLATOR WRITING SYSTEM

by

Jed B. Marti

Department of Comp. and Inf. Science
The University of Oregon
Eugene, Oregon 97403

ABSTRACT. The features of the Little Meta Translator Writing System are presented by the development of a compiler through three stages.

KEYWORDS. Translator Writing System, Compiler-Compiler, Pattern Matching.

A SESSION WITH THE LITTLE META TRANSLATOR WRITING SYSTEM

Little META is a LISP based Translator Writing System. It is a package of support routines and a complete translator designed to support the programming language experimentor. Little META is implemented in LISP, specifically the UOLISP subset [1] of Standard LISP [2].

The system has evolved continuously from one implemented by Loos [3]. A very fine implementation was done by Jenks [4]. A large version of the system called META/REDUCE has been used on a number of mainframes running Standard LISP [5]. A subset of Little META implemented by Kessler is part of the Portable Standard LISP project[6].

The system differs from conventional translator writing systems in several ways. It is implemented on a microcomputer and is small enough to fit easily within the storage constraints of such a system. It is implemented within the environment of an incremental programming system. Consequently translators are modifiable without complete recompilation. Thirdly, its input syntax closely resembles the context free grammar with semantic actions used in [7]. This feature is particularly useful in demonstrating compiler construction in the classroom situation. Except for a kernel of support routines, the translator is implemented in its own syntax. Modifications to the translator are very simple as its source is only three pages long. The system is portable to the extent that only trivial modifications will permit it to be moved to other versions of Standard LISP. Finally, the

translators and compilers it produces can run standalone in the LISP environment.

We will examine both the implementation of a compiler which produces Z80 assembly code and an interpreter of programs for the very simple Little Programming Language (LPL).

The Little META Source Language

Our strategy will be to first define the syntax and semantics of LPL, and then to implement it piecemeal and test it as we go along.

A Context Free Grammar for LPL.

LPL has a bare minimum of control constructs and no variable declarations. The semantics of the language hold no suprises and are the ones normally associated with the constructs.

1. <LPL program> -> <statement list> END;

2. <statement list> -> <statement>; <statement list> | ∈

3. <statement> -> <unlabelled statement> | <labelled statement>

4. <labelled statement> -> <identifier>: <unlabeled statement>

5. <unlabeled statement> ->
   <assignment statement> |
   <compound statement> |
   <conditional statement> |
   <transfer statement> |
   <input/output statement>

6. <assignment statement> ->
   LET <identifier> -> <arithmetic expression>

7. <arithmetic expression> -> <primary> <expression>

8. <expression> -> + <primary> <expression> |
   - <primary> <expression> | $\in$

9. <primary> -> <identifier> | <integer> |
   (<arithmetic expression>)

10. <compound statement> -> BEGIN <statement list> END

11. <conditional statement> -> IF <relational expression>
    THEN <unlabeled statement>

12. <relational expression> ->
    <arithmetic expression> <operator> <arithmetic expression>

13. <operator> -> = | < | >

14. <transfer statement> -> GO TO <identifier>

15. <identifier list> -> <identifier>, <identifier list> | $\in$

16. <input/output statement> -> INPUT <identifier list> |
    OUTPUT <identifier list>

There are two purposeful omissions from the BNF description, that of <integer> and <identifier>. We will assume that these are defined as simple unsigned integers and standard LISP style identifiers (any number of alphanumeric characters the first of which must be alphabetic, non-alphanumeric characters are prefixed with !).

The procedure we will follow in the development of the LPL compiler is to build three successive programs.

1. A syntax scanner. This program will verify that the syntax of the source program is correct.

2. The syntax scanner will be augmented to form an interpreter. This program will accept a complete LPL program and translate it into LISP for immediate execution.

3. The interpreter will be further augmented to implement the compiler. This program will accept LPL programs and produce Z80 assembly code.

The strategy in constructing each one of these programs will be to build the lowest level pieces first. This includes the arithmetic expressions, and the relational expressions. The next step will be to build the individual statements and then all the pieces will be drawn together to form the complete program. Working from the bottom up serves tutorial purposes only and is not the recommended way of implementing translators.

Little META Syntax

Just as a CFG description of syntax is made up of phrases, a Little META implementation of a programming language is made up of rules. A Little META rule is:

<META rule> -> <identifier>: <rule body> ;

That is, a <META rule> is an identifier followed by a colon followed by the rule body. The identifier is the name of the rule and the body

describes the syntax which the rule recognizes and actions to take when this occurs.

The <rule body> consists of one or more alternatives which are possible forms which the piece of language being described can take. These alternatives are separated by slashes (/). The alternatives are formed from tests which are different forms of syntax which must occur in the source language for the rule to succeed. One of the simplest tests is for terminal symbols. When a such a symbol is to occur in the source text, it is given in the Little META rule prefixed by an apostrophe ('). In Little META the <operator> phrase is implemented in the following fashion:

```
OPERATOR: '= / '< / '> ;
```

For the most part, Little META syntax analysis will be almost the same as the CFG syntax phrasing.

Tests can succeed or fail. A test succeeds when the source program contains an instance of the test at the position currently being scanned. A test fails if there is no such instance.

The last symbol in a program is a special case. Normally Little Meta parsers scan one token ahead. The last token of a program is not usually followed by another. So that an end of file condition does not occur, a special terminal symbol test is implemented, the final symbol prefixed by @.

In addition to the terminal symbol tests, a test can be the name of some other rule. The name of the rule appears without the brackets that surround the phrase name. Thus the <arithmetic expression> phrase is

coded:

```
ARITHMETIC-EXPRESSION: PRIMARY EXPRESSION;
```

That is, an ARITHMETIC-EXPRESSION is a PRIMARY followed by an EXPRESSION.

The structure underlying the alternative matching and concatenations of tests closely resembles the AND/OR programs of Harel [8].

Rather than try to implement a Little META rule which matches $\in$, a special syntax is implemented which permits a rule to succeed when none of its alternatives do. When a rule has one or more alternatives, one of which is $\in$ the alternatives are enclosed in brackets < ... >. Thus the <expression> phrase is implemented:

```
EXPRESSION: < '+ PRIMARY EXPRESSION / '- PRIMARY EXPRESSION>;
```

The $\in$ phrase does not occur. To paraphrase EXPRESSION: an EXPRESSION is either + followed by a PRIMARY and another EXPRESSION, or a - followed by a PRIMARY and another EXPRESSION. Note that EXPRESSION always succeeds no matter where the lexical scanner is in the current input string.

We can now define the entire <arithmetic expression> phrase set in META syntax:

```
ARITHMETIC-EXPRESSION: PRIMARY EXPRESSION;
EXPRESSION: < '+ PRIMARY EXPRESSION / '- PRIMARY EXPRESSION >;
PRIMARY: ID / NUM / '( ARITHMETIC-EXPRESSION ');
```

The ARITHMETIC-EXPRESSION program will produce a yes or error answer: yes the source string is a valid LPL arithmetic expression, or an error

message if it is not. The following are LPL expressions which ARITHMETIC-EXPRESSION will recognize:

```
VAR1
A + B
A - B + C
(A + B) - (C - (D + E))
```

The following expressions will not be recognized:

```
)A - B)
(a +)
-1 - -3
A * B
```

A task which frequently arises in the code of compilers and translators is the parsing of lists of things separated by punctuation marks. Two such forms occur in LPL, a list of identifiers separated by commas and the list of statements separated by semicolons. Little Meta provides a built in test for this task, the repetition construct. Its general form is:

```
<test>-<punctuation>-
```

where <test> is a test for the things which are to be repeated and <punctuation> is the punctuation mark which separates them. The repetition test succeeds when at least one <test> appears. If the <test> item is followed by the punctuation mark <punctuation> then another <test> must occur. Using repetition notation the <identifier list> phrase is implemented:

```
IDENTIFIER-LIST: ID-,- ;
```

We are now in a position to code the entire LPL syntax scanner. In the text of the scanner, comments are prefixed by % and run until the end

of the line. The rules of the LPL parser will exactly parallel those of the CFG description with annotation corresponding to the CFG phrase number.

```
(META 'LPL-SCANNER T)      % Invoke Little META Translator.

LPL-SCANNER: STATEMENT-LIST @END ;      % 1.

STATEMENT-LIST: STATEMENT-;- ;      %2.

STATEMENT: UNLABELLED-STATEMENT / LABELLED-STATEMENT ;      % 3.

LABELLED-STATEMENT: ID ': UNLABELLED-STATEMENT;      % 4.

UNLABELLED-STATEMENT: ASSIGNMENT-STATEMENT /      % 5.
    COMPOUND-STATEMENT /
    CONDITIONAL-STATEMENT /
    TRANSFER-STATEMENT /
    INPUT-OUTPUT-STATEMENT ;

ASSIGNMENT-STATEMENT: 'LET ID ':= ARITHMETIC-EXPRESSION ;      % 6.

ARITHMETIC-EXPRESSION: PRIMARY EXPRESSION ;      % 7.
EXPRESSION: < '+ PRIMARY EXPRESSION /      % 8.
    '- PRIMARY EXPRESSION > ;
PRIMARY: ID / NUM / '( ARITHMETIC-EXPRESSION ') ;      % 9.

COMPOUND-STATEMENT: 'BEGIN STATEMENT-LIST 'END ;      % 10.

CONDITIONAL-STATEMENT:      % 11.
    'IF RELATIONAL-EXPRESSION 'THEN UNLABELLED-STATEMENT ;
RELATIONAL-EXPRESSION:      % 12.
    ARITHMETIC-EXPRESSION OPERATOR ARITHMETIC-EXPRESSION ;
OPERATOR: '= / '< / '> ;      % 13.

TRANSFER-STATEMENT: 'GO 'TO ID ;      % 14.

IDENTIFIER-LIST: ID-,- ;      % 15.
INPUT-OUTPUT-STATEMENT:      % 16.
    'INPUT IDENTIFIER-LIST /
    'OUTPUT IDENTIFIER-LIST ;

FIN
```

The first line of the translator is the invocation of the Little Meta system. Little Meta is a large LISP program of which the main program is called META. It has two arguments, the first is the root rule of the translator being implemented. The second argument is T if the

translator is being defined, or NIL if this is a modification being made to an old one.

The LISP code resulting from the Little Meta translation of LPL-SCANNER can be executed with the help of the INVOKE function. INVOKE does the initialization required for the operation of a Little Meta generated program. The following sequence is an execution of the LPL-SCANNER syntax scanner. The LPL source program accepts two numbers from an input device and computes their quotient and remainder and displays them. Output from the system is in upper case, input lower case.

```
(invoke 'lpl-scanner)

 ENTERING LPL-SCANNER ...

    input dvdnd, dvsr;
    let q := 0;
loop: if (dvdnd - dvsr) < 0 then go to done;
    let q := q + 1;
    let dvdnd := dvdnd - dvsr;
    go to loop;
done: output q, dvdnd
    end

... EXITING LPL-SCANNER
```

A feature of Little Meta is that the translators it generates have built in error detection and automatic error message generation. For example, the LPL-SCANNER rule body has a STATEMENT-LIST followed by the terminal symbol END. If the STATEMENT-LIST is parsed correctly, but the END keyword is not found then an error message will be displayed and the translator will stop.

```
(invoke 'lpl-scanner)

ENTERING LPL-SCANNER ...

   input a;
   output a
   emd

***** ((DELIMITER END) LPL-SCANNER)
```

All error messages are prefixed with five asterisks. Note that the name of the rule in which the error was detected also appears. A different message is generated when the syntax error is detected during the repetition test. For example:

```
(invoke 'lpl-scanner)

 ENTERING LPL-SCANNER ...

   input a, b, ;

 ***** ((ID REPEATED SEPARATED BY ,) IDENTIFIER-LIST)
```

In this case, the missing item (an identifier) is named, what it is supposed to follow, and the rule name in which the error was detected. This is the most important reason for using descriptive rule names (some of which get very long) as they will be used in error messages. The messages are compiled from the structure of the test using a method closely allied to that of Hartmann described by Pemberton [9]. To keep the size of translators and the Little Meta system within limits the complete error recovery scheme has not yet been implemented.

The LPL Interpreter

The second program in the construction regime is an LPL interpreter. The strategy is to convert LPL programs into LISP and then to execute them using the LISP interpreter. One of the advantages of LISP over other languages used to implement translator writing systems is that creation of LISP programs by LISP programs and their immediate execution is well supported.

The LPL syntax scanner will be augmented to construct a LISP program as it scans the LPL source program. There are a number of features of Little Meta which are specifically applicable to the construction of intermediate LISP and prefix forms.

The Semantic Stack

A Little Meta rule considered as a LISP function returns T or NIL to indicate success or failure in recognition its construct in the source program. To communicate more than just success or failure, the semantic stack is used. When a rule succeeds it normally leaves one or more items on the top of this stack. Other rules can remove this information or use it to create larger expressions to be placed on the stack.

Any element on the stack can be accessed by its position relative to the top of the stack. Thus ##1 is the item on the top of the stack, ##2 the second element, ##3 the third and so on. There is no restriction on the size of the stack except the number of LISP free cells available at any given time. To access an item and remove it from the stack at the same time, the syntax is #1 for the first

element, #2 for the second and so on. These forms most commonly appear as part of the stack constructor form which has as a syntax:

```
<stack constructor> -> +<constructor item> |
    +(<constructor list>)
<constructor list> -> <constructor item><constructor list> |
    <nothing>
<constructor item> ->
    =<LISP S-expression> |
    -<constructor item> |
    ##<integer> |
    #<integer> |
    $<integer> |
    $<identifier> |
    <identifier> |
    <integer> |
    <string> |
    ( <constructor list> )
```

Rather than explain all of the forms they will be examined as they come into use in different parts of the translator.

The code for converting an LPL <arithmetic expression> into an executable LISP form is as follows:

```
ARITHMETIC-EXPRESSION: PRIMARY EXPRESSION;
EXPRESSION: <'+ PRIMARY +(PLUS #2 #1) EXPRESSION /
    '- PRIMARY +(DIFFERENCE #2 #1) EXPRESSION> ;
PRIMARY: ID / NUM / '( ARITHMETIC-EXPRESSION ') ;
```

The only difference between this rule set and that used in the syntax scanner are the two extra constructors in the EXPRESSION rule. Examining the rule set from PRIMARY upwards is the best way to understand the operation of ARITHMETIC-EXPRESSION. In PRIMARY, which ever of the three alternatives succeeds leaves an item on top of the semantic stack. The built in lexical rules ID and NUM leave their corresponding tokens on top of the stack. We will assume that the ARITHMETIC-EXPRESSION rule leaves a complete translated expression on top of the stack.

The EXPRESSION rule is tried only after the ARITHMETIC-EXPRESSION rule or after a previous use of EXPRESSION. When it is entered there is either a PRIMARY on top of the stack (from ARITHMETIC-EXPRESSION), or a completely translated EXPRESSION (from a recursive EXPRESSION call). If a + sign is found in the source string, and a PRIMARY is found there will be two items on top of the stack, the first being the last PRIMARY found, and the second what ever was there when EXPRESSION was entered. The constructor forms a new expression representing the sum of these two elements without evaluating them. The same is true if - is found, except that the DIFFERENCE of the two forms is constructed.

Note that, identifiers, numbers, and strings which occur without prefixes in constructors are copied as is into the constructed forms. References to the semantic stack either by # or ## are replaced by the values retrieved from the stack.

Let us trace the execution of this rule set on a LPL source string and watch the contents of the stack at various points.

At:	Source String and pointer	Stack
ARITHMETIC-EXPRESSION	A + B	<empty>
PRIMARY	A + B	<empty>
exit: PRIMARY	A + B	1:A
EXPRESSION	A + B	1:A
PRIMARY	A + B	1:A
exit:PRIMARY	A + B	1:B, 2:A
exit:EXPRESSION	A + B	(PLUS A B)
exit:ARITHMETIC-EXPRESSION	A + B	(PLUS A B)

Some LPL expressions and their translated equivalents are given in the following table.

LPL Expression	Translated Expression
VAR	VAR
VAR1 + VAR2 + VAR3	(PLUS (PLUS VAR1 VAR2) VAR3)
(A + B) - (C + D)	(DIFFERENCE (PLUS A B) (PLUS C D))
1 - BACK - TWOX	(DIFFERENCE (DIFFERENCE 1 BACK) TWOX)

To further illustrate the use of the # construct, consider the code for conditional expressions. The goal of the translation is to produce the LISP equivalent of an IF statement, this being the COND function. The first argument of the COND function is the relational expression, the second the statement to perform if the expression is true.

```
CONDITIONAL-STATEMENT: 'IF RELATIONAL-EXPRESSION 'THEN
    UNLABELLED-STATEMENT +(COND (#2 #1)) ;
RELATIONAL-EXPRESSION:
    ARITHMETIC-EXPRESSION RELATIONAL-OPERATOR ARITHMETIC-EXPRESSION
    +(#2 #2 #1) ;
RELATIONAL-OPERATOR: '= +EQUAL / '< +LESSP / '> +GREATERP ;
```

The RELATIONAL-OPERATOR rule succeeds on any of the defined relational operators, =, <, or >, and loads onto the top of the stack the LISP function name corresponding to the operator. The RELATIONAL-EXPRESSION rule parses two arithmetic expressions and the relational operator. Before the constructor is executed there are three items on the semantic stack: 1) the second ARITHMETIC-EXPRESSION, 2) the relational operator LISP function name, 3) the first arithmetic expression. The two #2's in the constructor reflect the fact that the execution of the first one causes the third item in the stack to become the new second item. In the CONDITIONAL-STATEMENT rule, two items are on the stack, the top element being the LISP form of the UNLABELLED-STATEMENT and the second being the LISP form of the RELATIONAL-EXPRESSION. These are combined into a complete LISP conditional. Note that the extra parentheses in the constructor actually become part of the created structure. The following IF statement:

```
IF (A + 12) < 0 THEN GO TO HELLO;
```

would be translated into:

```
(COND ((LESSP (PLUS A 12) 0) (GO HELLO)))
```

Repetition and the Semantic Stack

The repetition construct test-x- leaves a list of the items created by the test on the semantic stack. Thus the rule:

```
IDENTIFIER-LIST: ID-,- ;
```

leaves a list of the identifiers it found on top of the stack. The same is true for the STATEMENT-LIST function which will be defined later.

Often it is required that something be done with the list of items created by the repetition form. The FOR EACH construct implements actions on lists of things. Its format is:

```
<for each clause> ::=
    FOR EACH $<identifier> IN <expression-1>
        DO <expression-2>
```

The meaning of this clause is that <expression-2> is evaluated for each element of <expression-1> with the <identifier> being set to consecutive elements of <expression-1>. The <identifier> can occur in <expression-2>.

The $<identifier> form is a way of introducing local variables into rules. Local variables may occur almost anywhere, in constructors, LISP S-expressions, tests, and so on as long as they are prefixed by a dollar sign. The usual rules of LISP local variables apply to their use.

We combine these two constructs to implement the INPUT and OUTPUT statements in LPL. The INPUT statement is to be translated into a number of assignments to the variables of the list using the built in LISP READ function. For instance:

```
INPUT COFACT, BANGER;
```

will be translated to:

```
(SETQ COFACT (READ))
(SETQ BANGER (READ))
```

The OUTPUT statement will be converted to a number of calls on the LISP PRINT function. For example:

```
OUTPUT TWELVE, FREEP;
```

will be translated into:

```
(PRINT TWELVE)
(PRINT FREEP)
```

Using the code from the LPL syntax scanner as a guide the following rule set will generate the appropriate SETQ's and PRINT's.

```
IDENTIFIER-LIST: ID-,- ;  % 14. No change from LPL scanner.
INPUT-OUTPUT-STATEMENT:   % 15. Build READ's and PRINT's.
    'INPUT IDENTIFIER-LIST
        FOR EACH $X IN #1 DO +(SETQ $X (READ)) /
    'OUTPUT IDENTIFIER-LIST
        FOR EACH $X IN #1 DO +(PRINT $X) ;
```

The two FOR EACH clauses will repeatedly execute +(SETQ $X (READ)) and +(PRINT $X) to load items onto the stack. The original list of identifiers is removed. The -;- repetition that is used in STATEMENT-LIST will do the correct thing and not make all the generated LISP forms into one list.

It is often the case that we wish to concatentate two or more lists together before they are placed on the stack. This is the case of the LPL program. An LPL program will be built into a LISP PROG function which contains a list of all the variables used in the program, and all

the statements which are parsed into LISP by the individual statement routines. The constructor function + has a modifier which permits one list to be concatenated (appended) to another. A minus sign is prefixed to the form to be appended. If STATEMENT translates a single statement into a LISP form the rule XPROGRAM will build the proper PROG form out of it (ignoring variables for the moment).

```
XPROGRAM: STATEMENT-;- @END
     +(PROG NIL -#1) ;
```

Remember that the -;- construct builds a list of STATEMENTs, this list is appended to the (PROG NIL) list before it. Thus if ((SETQ A 12) (SETQ B 34)) was on the stack, the PROG loaded onto the stack will be (PROG NIL (SETQ A 12) (SETQ B 34)) and <u>not</u> (PROG NIL ((SETQ A 12) (SETQ B 34))) which would be the case if the - prefix was not used.

In order for the variables used in the LPL program to be collected and placed into the PROG form, a global variable will accumulate them when they occur on the left side of an assignment statement or in an INPUT statement. This global variable must be declared before its use. Little Meta permits any LISP function to be called by placing a period in front of the form and a semicolon after it. This construct is used in place of what would normally be a rule.

The value of a variable or <u>any</u> LISP expression can be obtained by placing an = sign in front of the expression. This very powerful feature permits you to use LISP when the Little Meta syntax is not complete enough to accomplish some task. To place the list of variables in the PROG form, the global variable name prefixed by = is placed in the + constructor form. Likewise, the = prefix can be used to convert any LISP function into a test.

The dot prefix (.) causes an expression to be evaluated, but its result to be ignored. This form can be used in rules much in the same way as a simple test, but one that always succeeds. In the LPL-INTERPRETER rule this form will be used to initialize the global variable which will contain the list of variables used in the final constructed PROG form. It will also be used to cause the evaluation of the constructed PROG. The complete LPL-INTERPRETER rule and associated functions looks like this:

```
.(GLOBAL '(VARIABLES));                    % 1.
LPL-INTERPRETER: .(SETQ VARIABLES NIL)     % 2.
    STATEMENT-LIST-;- @END                 % 3.
    +(PROG =VARIABLES -#1)                 % 4.
    .(PRINT ##1)                           % 5.
    .(EVAL #1) ;                           % 6.
STATEMENT-LIST: STATEMENT-;- ;             % 7.
```

Line 1 is the declaration of the global variable VARIABLES. The first line of the LPL-INTERPRETER rule, line 2, causes this "symbol table" to be emptied before the LPL program is translated. The dot prefix causes the result of this assignment to be ignored (remember that NIL means failure and NIL is the value of this clause). Line 3 invokes STATEMENT-LIST repeatedly as long as there are semicolons between statements and until the first keyword of a statement is END. All the statements so parsed are combined into one list and placed on the top of the stack. Line 4 builds the final PROG form to be evaluated. The list of LPL variables will have been placed in rules. The - prefix on #1 causes the list of statements to be appended to the list of two elements (PROG VARIABLES). Since VARIABLES will already be a list there is no need to enclose it in parentheses. Line 5 causes the PROG form to be displayed. This is just a trace feature so that we can determine that the correct internal form of the LPL-program was created. Line 6

causes the evaluation of the form. This evaluation is the execution of the LPL program. Line 7 is the definition of a list of statements.

With the addition of a UNION function the entire LPL interpreter can be defined. The UNION function is used to add variables to the VARIABLES list so that no variable will appear more than once.

```
(META 'LPL-INTERPRETER T)        % Invoke Little META Translator.

.(GLOBAL '(VARIABLES));          % Declare global variable.

.(DE UNION (A B)      % Define UNION function in LISP.
    (COND ((NULL A) B)
          ((MEMQ (CAR A) B) (UNION (CDR A) B))
          (T (CONS (CAR A) (UNION (CDR A) B))))) ;

LPL-INTERPRETER: .(SETQ VARIABLES NIL)      % 1.
    STATEMENT-LIST @END
    +(PROG =VARIABLES -#1)
    .(PRINT ##1)
    .(EVAL #1) ;

STATEMENT-LIST: STATEMENT-;- ;      % 2.

STATEMENT: UNLABELLED-STATEMENT / LABELLED-STATEMENT ;      % 3.

LABELLED-STATEMENT: ID ': UNLABELLED-STATEMENT ;      % 4.

UNLABELLED-STATEMENT: ASSIGNMENT-STATEMENT /      % 5.
    COMPOUND-STATEMENT /
    CONDITIONAL-STATEMENT /
    TRANSFER-STATEMENT /
    INPUT-OUTPUT-STATEMENT ;

ASSIGNMENT-STATEMENT: 'LET ID ':=      %6.
    .(SETQ VARIABLES (UNION (LIST ##1) VARIABLES))
    ARITHMETIC-EXPRESSION
    +(SETQ #2 #1) ;

ARITHMETIC-EXPRESSION: PRIMARY EXPRESSION ;      % 7.
EXPRESSION: < '+ PRIMARY +(PLUS #2 #1) EXPRESSION /      % 8.
    '- PRIMARY +(DIFFERENCE #2 #1) EXPRESSION > ;
PRIMARY: ID / NUM / '( ARITHMETIC-EXPRESSION ') ;      % 9.

COMPOUND-STATEMENT: 'BEGIN STATEMENT-LIST 'END
    +(PROG NIL -#1) ;      % 10.

CONDITIONAL-STATEMENT:      % 11.
    'IF RELATIONAL-EXPRESSION 'THEN UNLABELLED-STATEMENT'
    +(COND (#2 #1)) ;
RELATIONAL-EXPRESSION:      % 12.
```

```
    ARITHMETIC-EXPRESSION OPERATOR ARITHMETIC-EXPRESSION
    +(#2 #2 #1) ;
OPERATOR: '= +EQUAL / '< +LESSP / '> +GREATERP ;      % 13.

TRANSFER-STATEMENT: 'GO 'TO ID +(GO #1) ;      % 14.

IDENTIFIER-LIST: ID-,- ;      % 15.
INPUT-OUTPUT-STATEMENT:       % 16.
    'INPUT IDENTIFIER-LIST
      .(SETQ VARIABLES (UNION ##1 VARIABLES))
      FOR EACH $X IN #1 COLLECT +(SETQ $X (READ)) /
    'OUTPUT IDENTIFIER-LIST
      FOR EACH $X IN #1 COLLECT +(PRINT $X) ;

FIN
```

Let us now try the demonstration LPL program. We will divide the number 15 by the number 6 which should give us 2 with remainder 3 as the output.

```
(invoke 'lpl-interpreter)

 ENTERING LPL-INTERPRETER ...

    input dvdnd, dvsr;
    let q := 0;
loop: if (dvdnd - dvsr) < 0 then go to done;
    let q := q + 1;
    let dvdnd := dvdnd - dvsr;
    go to loop;
done: output q, dvdnd
    end

(PROG (DVDND DVSR Q)
      (SETQ DVDND (READ))
      (SETQ DVSR (READ))
      (SETQ Q 0)
LOOP (COND ((LESSP (DIFFERENCE DVDND DVSR) 0) (GO DONE)))
      (SETQ Q (PLUS Q 1))
      (SETQ DVDND (DIFFERENCE DVDND DVSR))
      (GO LOOP)
DONE (PRINT Q)
      (PRINT DVDND))

15        <- input.
6

2
3

... EXITING LPL-INTERPRETER
```

The first output following the LPL-source program is the translated LISP version which will be interpreted. The execution of the program follows. The dividend and the divisor are read in and the quotient and remainder are printed out. Notice that division by zero will manifest itself as an infinite loop and that division where large quotients are involved is none too efficient.

The LPL Compiler.

The organization of the LPL compiler will follow very closely the template provided by the LPL syntax scanner and interpreter. The compiler we are constructing will in a single pass produce Z80 assembly code and storage locations for the variables. This code will be dumped to a disk file which when concatenated to a library of I/O routines can be assembled and executed.

The code produced will be symbolic Z80 instructions in the Zilog format [10]. All storage locations, registers, and arithmetic will use 16 bit quantities.

Formatted Output

Rather than output the numerical values of Z80 machine instructions, assembly code will be generated as character strings. This indirect approach has a number of advantages:

1. If need be, the code can be edited and optimized by hand.

2. It is easier to debug the compiler code generators.

3. We don't have to keep track of addresses or know the size of

machine instructions.

4. We don't have to write an assembler.

The assembly output will be produced with the help of the Little Meta formatted output routines. An output clause has the following structure:

```
=>(item[0], ..., item[1])
```

where the items are any of the following:

COL:n - causes the output pointer to skip to column n.

SPACE:n - causes n spaces to be skipped.

/ - causes a skip to a new line.

Any LISP Expression - The value of the expression is displayed by the LISP PRIN2 function.

Perhaps the simplest form to generate code for is the GO TO statement. Once the GO, TO, and identifier have been parsed, a Z80 long jump instruction will be generated with its destination address the label name.

```
TRANSFER-STATEMENT:
    'GO 'TO ID
    =>(COL:7, "JP", COL:15, #1, /) ;
```

The instruction and operands are in fixed columns for cleaner output. There is no check made that the label is defined, multiply defined, or a variable name. Any such checking will be left to the assembler. It is not our purpose to implement a production compiler, rather an

experimental one. Once the language features are fixed, the compiler can be implemented in its own syntax and include more detailed error checking.

For the input and output statements there will be two library routines, READ and PRINT. READ will accept a single 16 bit numeric value from some I/O device and return it in the HL register pair. PRINT takes a value in the 16 bit register HL and displays it on some I/O device. The implementation of the compiler INPUT-OUTPUT-STATEMENT parallels that of the interpreter rule.

```
INPUT-OUTPUT-STATEMENT: 'INPUT IDENTIFIER-LIST
    .(SETQ VARIABLES (UNION ##1 VARIABLES))
    FOR EACH $X IN #1 COLLECT
      =>(COL:7, "CALL", COL:15, "READ", /,
         COL:7, "LD", COL:15, "(", $X, "),HL", /) /
    'OUTPUT IDENTIFIER-LIST
    FOR EACH $X IN #1 COLLECT
      =>(COL:7, "LD", COL:15, "HL,(", $X, ")", /,
         COL:7, "CALL", COL:15, "PRINT", /) ;
```

Notice that the / inside the output clause is different than the / used to separate the two alternative. Each variable in the list parsed generates two separate Z80 instructions.

The Pattern Matcher

Rather than generating code during the parsing of arithmetic expressions, the expression will first be converted into the prefix form (LISP S-expressions) and then these will be matched against various patterns to produce different output code sequences. We now describe the patterns and actions in relation to the preparation of arithmetic expression assembly language code.

The advantages of using patterns over direct generation of code

during parsing are not clear. Since it does separate code generation from parsing, each of these pieces is easier to complete as a programming task and can be tested in isolation from the other. Our experience has been that writing complex patterns based on the parse tree is much easier than including code generation in the parser. Much of this code replicates what the pattern matcher provides for us already.

A pattern sequence is a set of patterns which are matched in order against a single LISP S-expression for both structure and content. The syntax of a pattern sequence is as follows:

```
<pattern name> =
    <pattern[0]> -> <action[0]> ,
    <pattern[1]> -> <action[1]> ,
    .
    .
    .
    <pattern[n]> -> <action[n]> ;
```

The object of a pattern sequence is to match a single S-expression against the patterns one at at time until one match succeeds. Then the corresponding action is taken.

A pattern is a template against which the actual parameter of the pattern sequence is matched. Patterns are either atomic entities or expressions formed from pattern primitives.

Occurrences of atoms in a pattern must exactly match the source against which the pattern is being matched. Thus the pattern:

```
(NOW WE ARE 6) -> ...
```

will match only the list (NEW WE ARE 6) and no other.

To match an arbitrary S-expression, the &n construct is used. 'n' is an integer from 1 to 4095 which serves to identify this particular expression. Thus:

```
(NOW WE ARE &1) ->
```

will match any S-expression which has as its first elements NOW, WE, ARE in that order, and any LISP expression as its last element. Thus (NOW WE ARE 1), (NOW WE ARE GONE), and (NOW WE ARE (IN A LIST)) will all succeed when matched against this pattern. Furthermore, the piece of the expression which corresponds to the &n will be available on the action side.

Then &n can be made more selective by allowing it to match an expression only if the value of some function in relation to the expression is true. The form is:

```
&<(predicate)>n
```

where <predicate> is an expression which returns NIL or not based on the expression being matched. Thus to match only numbers, the LISP predicate NUMBERP is used:

```
&<(NUMBERP &1)>1
```

This pattern succeeds only if its corresponding expression is a number. The &1 inside the expression corresponds to the piece of the expression to be matched.

The action side of a pattern is executed when its antecedent is successfully matched to the source expression. The action side is a list of forms which at the top level are any of the following:

1. The stack reference and access functions # and ##.

2. Quoted LISP S-expressions.

3. Expressions prefixed with = for evaluation.

4. $ prefix local variables.

5. & pattern pieces.

6. Atoms.

7. Combinations of the above in expressions.

A very simple implementation of the patterns to generate code from the LISP form of the LPL arithmetic expression is as follows:

```
ARITHMETIC-CODE =

 &<(NUMBERP &1)> 1 -> =>(COL:7, "LD", COL:15, "HL,", &1, /),   % 1.

 &<(IDP &1)>1 -> =>(COL:7, "LD", COL:15, "HL,(", &1, ")", /), % 2.

 (PLUS &1 &2) -> =(ARITHMETIC-CODE &2)                          % 3.
    =>(COL:7, "PUSH", COL:15, "HL", /)
    =(ARITHMETIC-CODE &1)
    =>(COL:7, "POP", COL:15, "DE", /,
       COL:7, "ADD", COL:15, "HL,DE", /),

 (DIFFERENCE &1 &2) -> =(ARITHMETIC-CODE &2)                    % 4.
    =>(COL:7, "PUSH", COL:15, "HL", /)
    =(ARITHMETIC-CODE &1)
    =>(COL:7, "POP", COL:15, "DE", /,
       COL:7, "OR", COL:15, "A", /,
       COL:7, "SBC", COL:15, "HL,DE", /) ;
```

This very simple pattern sequence generates correct though inefficient code for arithmetic expressions with addition and subtraction. Patterns 1 and 2 work on the primitive values and output code to load register HL with integer values or variable locations. Rules 3 and 4 work on forms constructed from the primitives and generate the code to do 16

bit addition and subtraction.

The LPL expression

```
A + B - 12
```

generates the following intermediate LISP form:

```
(PLUS A (DIFFERENCE B 12))
```

We can follow this form through the patterns and actions.

Expression being matched	Matches Rule No.	Pieces /values	Output
1. (PLUS A (DIFFERENCE B 12))	3	&1:A, &2:(DIFFERENCE B 12)	

Rule number 3 has two recursive matches to be made, first on &2, and then on &1 to generate code for the two operands of PLUS.

2. (DIFFERENCE B 12)	4	&1:B, &2:12	

Rule 4 also has two recursive matches to be made. Only when the first has completed will some code be output.

3. 12	1	&1:12	LD HL,12

Rule 1 outputs code immediately. This completes the first recursion from match number 2 above. Some code is output now and the second recursion of part 2 commences.

```
PUSH HL
```

4. B	2	&1:B	LD HL,(B)

Both operands of the DIFFERENCE have been generated. The second argument is on the stack, the first is in HL. The code for the 16 bit subtraction is now generated. Note that all actions generate code which leaves their results in register HL.

```
POP  DE
OR   A
SBC  HL,DE
```

At this point the first recursion of match 1 has completed. The second now commences after the result of the first is pushed onto the stack.

```
PUSH HL
```

5. A	2	&1:A	LD HL,(A)

The final recursive match has completed and the addition can be completed.

```
POP  DE
ADD  HL,DE
```

To make the code generation more efficient requires a few more patterns and code output sequences. For instance one could have the pattern:

```
(PLUS &<(NUMBERP &1)>1 &<(NUMBERP &2)>2) ->
    =>(COL:7, "LD", COL:15, "HL,", =(PLUS &1 &2), /) ,
```

This pattern recognizes the sum of two numbers. Rather than output code to compute the sum, they are added at "compile time" and their sum is placed in register HL.

Occasionally it is necessary to generate local labels or identifiers, those which are used within a translated program but are not seen by the user. Within a Little Meta rule the $n construct causes automatic generation of a label which is guaranteed to be unique. The label associated with the number will remain constant during a single execution of the rule but will change for every new execution. The code generated for the conditional statement requires a label at the end of the code for the statement which is the object of a jump instruction. The jump will be made if the relational expression is <u>not</u> true. The complete code for the conditional statement is as follows:

```
CONDITIONAL-STATEMENT: 'IF RELATIONAL-EXPRESSION 'THEN
    =>(COL:7, "JP", COL:15, #1, ",", $1, /)
    UNLABELLED-STATEMENT
    =>($1, COL:7, "EQU", COL:15, "$", /) ;

RELATIONAL-EXPRESSION
    ARITHMETIC-EXPRESSION =(ARITHMETIC-CODE #1)
    =>(COL:7, "PUSH", COL:15, "HL", /)
    RELATIONAL-OPERATOR
    ARITHMETIC-EXPRESSION =(ARITHMETIC-CODE #1)
    =>(COL:7, "POP", COL:15, "DE", /,
       COL:7, "OR", COL:15, "A", /,
       COL:7, "SBC", COL:15, "HL,DE", /) ;

RELATIONAL-OPERATOR: '= +NZ / '< +M / '> +P ;
```

The RELATIONAL-EXPRESSION rule leaves on top of the stack, the condition code for the jump statement. The label on the jump statement is the same one as will be generated on the EQU after the UNLABELLED-STATEMENT code.

An alternative way of compiling the RELATIONAL-EXPRESSION code would be to generate the LISP expressions in the interpreter and then use a pattern to create code. With appropriate patterns this approach will generate much better code.

The entire compiler can now be implemented.

```
(META 'LPL!-COMPILER T)

.(GLOBAL '(VARIABLES)) ;

.(DE UNION (A B)
    (COND ((NULL A) B)
          ((MEMQ (CAR A) B) (UNION (CDR A) B))
          (T (CONS (CAR A) (UNION (CDR A) B))))) ;

LPL-COMPILER: .(SETQ VARIABLES NIL)     % 1.
    =>($1, COL:7, "EQU", COL:15, "$", /)
    STATEMENT-LIST @END
    =>(COL:7, "HLT", /)
    FOR EACH $X IN VARIABLES DO
      =>($X, COL:7, "DW", COL:15, "0", /)
    =>(COL:7, "END", COL:15, $1, /) ;

STATEMENT-LIST: STATEMENT-;- ;     % 2.
```

```
STATEMENT: UNLABELLED-STATEMENT / LABELLED-STATEMENT ;      % 3.

LABELLED-STATEMENT: ID ':      % 4.
    =>(#1, COL:7, "EQU", COL:15, "$", /)
    UNLABELLED-STATEMENT ;

UNLABELLED-STATEMENT: ASSIGNMENT-STATEMENT /      % 5.
    COMPOUND-STATEMENT /
    CONDITIONAL-STATEMENT /
    TRANSFER-STATEMENT /
    INPUT-OUTPUT-STATEMENT ;

ASSIGNMENT-STATEMENT: 'LET ID      % 6.
    .(SETQ VARIABLES (UNION (LIST ##1) VARIABLES))
    ARITHMETIC-EXPRESSION =(ARITHMETIC-CODE #1)
    =>(COL:7, "LD", COL:15, "(", #1, "),HL", /) ;

ARITHMETIC-EXPRESSION: PRIMARY EXPRESSION ;      % 7.
EXPRESSION: < '+ PRIMARY +(PLUS #2 #1) EXPRESSION /      % 8.
    '- PRIMARY +(DIFFERENCE #2 #1) EXPRESSION > ;
PRIMARY: ID / NUM / '( ARITHMETIC-EXPRESSION ') ;      % 9.

ARITHMETIC-CODE =      % Patterns for arithmetic code generation.
 &<(NUMBERP &1)>1 -> =>(COL:7, "LD", COL:15, "HL,", &1, /),
 &<(IDP &1)>1 -> =>(COL:7, "LD", COL:15, "HL,(", &1, ")", /),
 (PLUS &1 &2) -> =(ARITHMETIC-CODE &2)
    =>(COL:7, "PUSH", COL:15, "HL", /)
    =(ARITHMETIC-CODE &1)
    =>(COL:7, "POP", COL:15, "DE", /,
       COL:7, "ADD", COL:15, "HL,DE", /),
 (DIFFERENCE &1 &2) -> =(ARITHMETIC-CODE &2)
    =>(COL:7, "PUSH", COL:15, "HL", /)
    =(ARITHMETIC-CODE &1)
    =>(COL:7, "POP", COL:15, "DE", /,
       COL:7, "OR", COL:15, "A", /,
       COL:7, "SBC", COL:15, "HL,DE", /) ;

COMPOUND-STATEMENT: 'BEGIN STATEMENT-LIST 'END ;      % 10.

CONDITIONAL-STATEMENT: 'IF RELATIONAL-EXPRESSION      % 11.
    'THEN
    =>(COL:7, "JP", COL:15, #1, ",", $1, /)
    UNLABELLED-STATEMENT
    =>($1, COL:7, "EQU", COL:15, "$", /) ;
RELATIONAL-EXPRESSION: ARITHMETIC-EXPRESSION      % 12.
    =(ARITHMETIC-CODE #1)
    =>(COL:7, "PUSH", COL:15, "HL", /)
    RELATIONAL-OPERATOR
    ARITHMETIC-EXPRESSION
    =(ARITHMETIC-CODE #1)
    =>(COL:7, "POP", COL:15, "DE", /,
       COL:7, "OR", COL:15, "A", /,
       COL:7, "SBC", COL:15, "HL,DE", /) ;
RELATIONAL-OPERATOR: '= +NZ / '< +M / '> +P ;      % 13.

TRANSFER-STATEMENT: 'GO 'TO ID      % 14.
```

```
    =>(COL:7, "JP", COL:15, #1, /) ;

IDENTIFIER-LIST: ID-,- ;      % 15.
INPUT-OUTPUT-STATEMENT: 'INPUT IDENTIFIER-LIST      % 16.
    .(SETQ VARIABLES (UNION ##1 VARIABLES))
    FOR EACH $X IN #1 DO
      =>(COL:7, "CALL", COL:15, "READ", /,
         COL:7, "LD", COL:15, "(", $X, "),HL", /) /
    'OUTPUT IDENTIFIER-LIST
    FOR EACH $X IN #1 DO
      =>(COL:7, "LD", COL:15, "HL,(", $X, ")", /,
         COL:7, "CALL", COL:15, "PRINT", /) ;

FIN
```

The simple division program will generate the following code. The statements are listed with the code that they generate to help follow the compilation process.

```
(invoke 'lpl-compiler)

ENTERING LPL-COMPILER ...

G0001  EQU     $

  input dvdnd, dvsr;
        CALL    READ
        LD      (DVDND),HL
        CALL    READ
        LD      (DVSR),HL

    let q := 0;

        LD      HL,0
        LD      (Q),HL

loop: if (dvdnd - dvsr) < 0 then go to done;

LOOP    EQU     $
        LD      HL,(DVSR)
        PUSH    HL
        LD      HL,(DVDND)
        POP     DE
        OR      A
        SBC     HL,DE
        PUSH    HL
        LD      HL,0
        POP     DE
        OR      A
        SBC     HL,DE
        JP      M,G0002
        JP      DONE
```

```
G0002  EQU     $

    let q := q + 1;

        LD      HL,1
        PUSH    HL
        LD      HL,(Q)
        POP     DE
        ADD     HL,DE
        LD      (Q),HL

    let dvdnd := dvdnd - dvsr;

        LD      HL,(DVSR)
        PUSH    HL
        LD      HL,(DVDND)
        POP     DE
        OR      A
        SBC     HL,DE
        LD      (DVDND),HL

    go to loop;

        JP      LOOP

done: output q, dvdnd

DONE    EQU     $
        LD      HL,(Q)
        CALL    PRINT
        LD      HL,(DVDND)
        CALL    PRINT

    end

        HLT
Q       DW      0
DVSR    DW      0
DVDND   DW      0
        END     G0001

... EXITING LPL-COMPILER
```

Why Use a Translator Writing System

One of the problems with Translator Writing Systems is that they are generally slow and produce compilers that are not much faster. Though there is considerable research being carried out to improve this situation, translators and compilers produced by Little Meta are fairly slow. The place of Little Meta in the overall scheme of things is as an initial implementation tool. The steps that are usually followed run somewhat like this.

1. A language is defined in some form or other.

2. An interpreter is built for the language using Little Meta.

3. Many programs implemented in the language are implemented and debugged using the interpreter. During this process new features are added, removed, and old ones clarified as the need arises.

4. A compiler is implemented in Little Meta. The code produced runs stand alone or with a library on the host machine.

5. A compiler is implemented in the new language to run stand alone.

6. Little Meta compiles this compiler. The new compiler is capable of compiling itself and will probably run much faster. The Little Meta compiler and translator can be discarded. Certainly LPL is not sufficient as it stands to implement a compiler, but a few very simple modifications would provide enough of a language for this to be accomplished. The PILOT language is a well done example of this kind of project [11]. Its compiler implemented in PILOT is only a few hundred lines long and can be implemented in a week

or so.

This process is a particularly useful way of bringing up a compiler for a new machine which has few utilities and no readily available compilers of its own. With a machine capable of running Little Meta it is possible to bring up large amounts of software on a machine which has none.

Meta in Meta

Just as compilers for a language are implemented in the language, Little Meta is implemented in Little Meta. This process has been going on for a number of years with the first version hand implemented a long time ago. The system consists of 3 parts.

1. A support package to make Little Meta run. This consists of the lexical scanner, lexical primitives, parsing control, semantic stack operations, and error control.

2. A support package for Little Meta produced translators. This includes the table set up, symbol table primitives, formatted output, and the pattern matcher.

3. The Little Meta translator. This section of code is implemented in Meta itself and converts the Little Meta syntax into LISP object code. This code is only 3 pages long.

The total length of code is less than seven hundred lines including comments.

More Features

There are a number of other facilities provided by the system which were not covered in this report. These include the ability to "back up" the lexical scan when a rule only partially succeeds so that some other rule can be tried; a complete block structured symbol table package; complex patterns; the ability to interface to LISP and packages; and finally the ability to interact and modify a translator in a piecemeal fashion.

This last feature separates Little Meta from most non-LISP based translator writing systems. It is particularly attractive during the debugging of a translator. The suspect rule can be traced in the usual LISP fashion and then redefined if it is in error all without retranslating the entire rule set. One can stop the translator at various points and examine the semantic stack, global and local variables, the symbol table, and other state information.

Conclusions

Little Meta and is an attractive program development tool for microcomputers. The examples presented in this paper were run on a microcomputer with only 48k bytes of main storage. The compilation of the division program took less than one minute. The translation and compilation of the LPL-COMPILER took less than 3 minutes. To date the system has been used to:

1. Experiment with various syntaxes for a computer graphics programming language.

2. Compile itself.

3. Help students master the basics of compiler implementation.

The immediate projected uses for the system include:

1. To build cross compilers for the newer 16 bit microcomputer systems.

2. To build a special purpose language for implementing software for a local computer network.

Acknowledgements

The author would like to thank, Rudiger Loos, Richard Jenks, Cedric Griss, Robert Keller, and most of all Martin Griss and Robert Kessler for implementing the pattern matcher.

List of References

1. J. Marti, 'LISP for the TRS-80', to appear in 80 Microcomputing.

2. J. Marti, A. C. Hearn, M. L. Griss, C. Griss, 'Standard LISP Report' SIGPLAN Notices, 14 10 48-68 (October 1979).

3. R. Loos, private communication.

4. R. D. Jenks, 'META/LISP: an interactive translator writing system' IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, New York.

5. J. Marti, 'The META/REDUCE translator writing system' SIGPLAN Notices, 13 10 (October 1978) 42-49.

6. Utah Symbolic Computation Group, 'The portable Standard LISP users manual', Department of Computer Science, University of Utah, Salt Lake City, Utah, January 1982.

7. A. V. Aho and J. D. Ullman, 'Principles of compiler design', Addison-Wesley Publishing Company, 1977.

8. D. Harel, 'And/or programs: a new approach to structured programming', ACM TOPLAS, 2 1 1-17 (January 1980).

9. S. Pemberton, 'Comments on an error-recovery scheme by Hartmann', Software-Practice and Experience, 10 231-240 (1980).

10. "Z80-CPU Technical Manual", Zilog, Los Altos, California, 1976.

11. M. H. Halstead, 'A laboratory manual for compiler and operating system implementation', American Elsevier Publishing Company, Inc., New York, 1974.