December 1981

CIS-TR-81-10

AN INTRODUCTION TO 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.

AN INTRODUCTION TO THE LITTLE META TRANSLATOR WRITING SYSTEM

Little META is a LISP based Translator Writing System. A Translator Writing System (TWS for short) is a program or collection of routines which aid the development of translators; programs which convert programs in one source language to another. The microcomputer user might use a TWS for the following reasons:

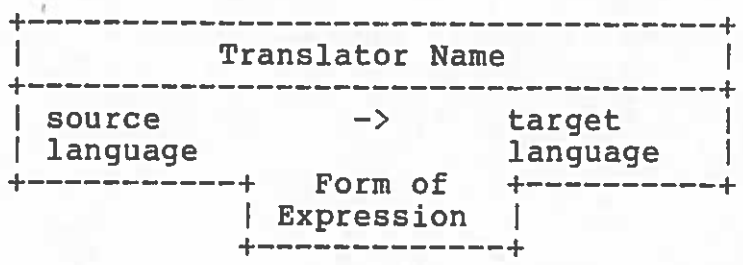
1. He is experimenting with programming languages and would like to develop a new one.

2. He would like to build a preprocessor for his favorite compiler to support programming constructs it does not support.

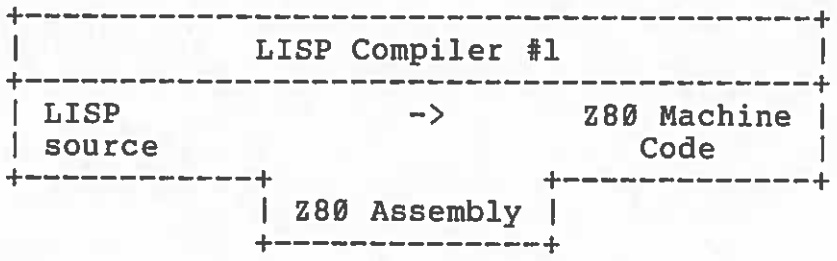
3. He would like to "front end" a complicated program with a translator which will accept programming language style input.

Little META is a package of support routines and a complete translator designed to support these activities. Little META is implemented in the LISP programming language, specifically the UOLISP variant previously described previously [1]. A manual describing all of its features and its use has been prepared [2].

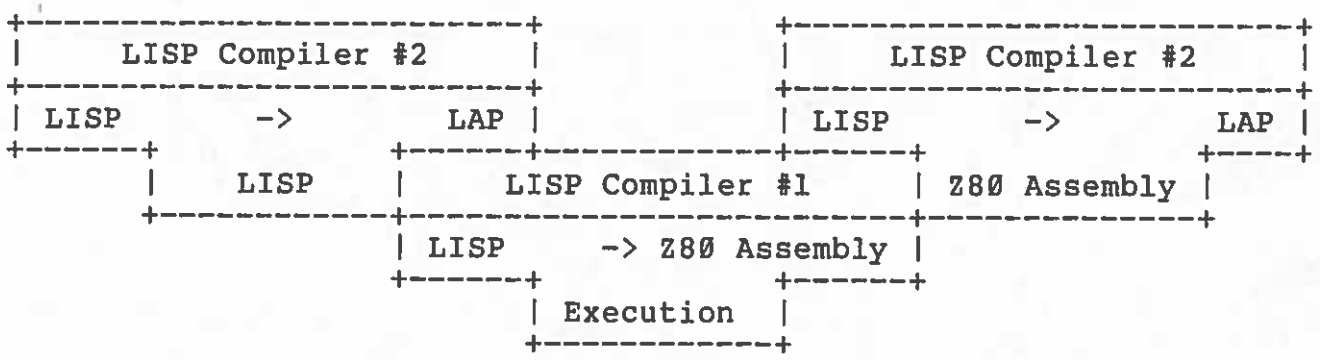
To fully appreciate what Little META does one needs to know the various translations which are taking place. Computer Scientists have developed a notation which helps to visualize the translation process [3].



This T diagram depicts the function of a translator which takes programs in the "source language" and transforms them into programs in the "target language". The "Form of Expression" indicates the form (or syntax) of the source of the translator program. For example, the compilation of LISP programs by a LISP compiler written in Z80 assembly code into executable Z80 instructions would be represented by:

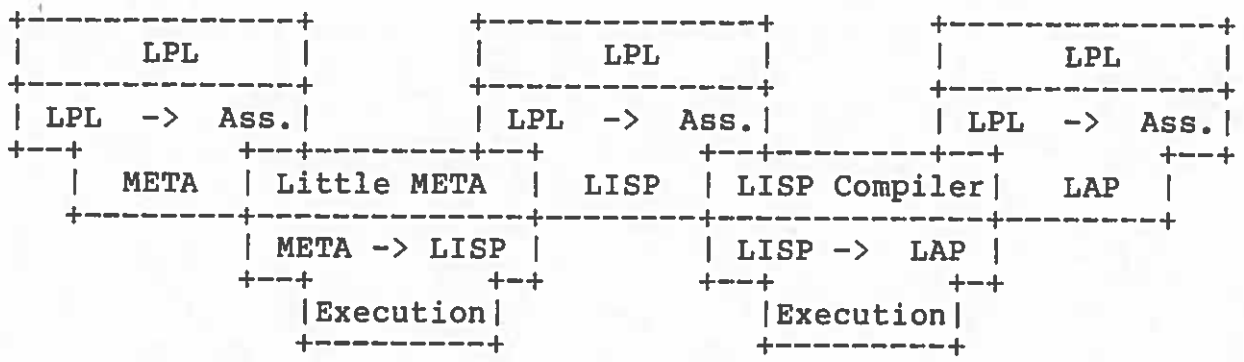


This diagram states that LISP Compiler #1 translates LISP source code into Z80 machine code. It also says that LISP Compiler #1 is implemented in Z80 Assembly language. When two or more translations of a program occur, a composite T diagram is used. Suppose that we have written LISP Compiler #2 in LISP which produces LISP Assembly Code (LAP). In order to get LISP Compiler #2 running we need LISP Compiler #1.



This composite diagram depicts the steps necessary to get LISP Compiler #2 into a form which can be executed. This is only possible with the assistance of LISP Compiler #1. Later on we will show how to "bootstrap" compilers using Little META.

The Little META system provides the basis for easy implementation of compilers and interpreters. To understand how this works we will examine the implementation of LPL (Little Programming Language). LPL will have both a compiler which produces Z80 assembly code, and an interpreter so that programs can be run without compilation. There are two programs involved in the implementation of LPL, the Little META system which converts the compiler description into LISP, and the LISP compiler which converts LISP into executable code, in this case LAP. Our task is the implementation of the LPL compiler in the syntax accepted by the Little META system. The T diagram which defines the process whereby we get an executable LPL compiler or interpreter from our META implementation of LPL is given in the following T diagram.



Simply stated, Little META translates the META description of LPL into LISP. The LISP version of LPL is then translated by the LISP compiler into LAP giving the LAP version of LPL which can be executed by the Z8Ø.

T diagrams might be seen as a lot of wasted effort for what on the surface is a very simple translation process. The clincher is that Little META is implemented in Little META. It has evolved to its present form in a near continuous process since its first implementation in 197Ø or there abouts. This process will be described later with the process of modifying Little META to suit your own needs. To understand how this occurred and how modifications can be made is an interesting study.

The Little META Source Language

We are now ready to approach the implementation of compilers and interpreters using the Little META syntax. 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.

BNF

The syntax of a programming language is the form of commands, statements, expressions and so on that the language compiler or translator can understand. Code which does not conform to the syntax of a language causes a syntax error to be detected by the compiler. To define the syntax of a programming language a meta language called BNF (Backus Nauer Form) was invented. It was first used to describe ALGOL 60 [4]. BNF is extraordinarily simple. The syntax of a programming language is defined by a number of named phrases. Things like: program, if-statement, arithmetic expression, end-statement, variable, integer and the like are phrases. Phrase names are enclosed in brackets as in <program> or <if-statement>. If a particular phrase can have more than one possible syntax, its alternatives are separated by vertical bars (|). A complete phrase in a BNF description is a phrase name followed by ::= followed by the names of other phrases of which the phrase is composed. A phrase can also have terminal symbols. Terminal symbols appear as is in the source language without the surrounding brackets. Thus the keyword IF in an IF...THEN...ELSE statement is a terminal symbol as are THEN and ELSE.

A BNF Example.

To make it fairly simple LPL is going to support only addition and subtraction of integers. It will also permit parentheses to be used to make the arithmetic expressions easier to read. To define the syntax of arithmetic expressions in this form requires three phrases:

```
<arithmetic expression> ::= <primary> <expression>
<expression> ::= + <primary> <expression> |
                 - <primary> <expression> | <nothing>
```

```
<primary> ::= <identifier> | <integer> |
              (<arithmetic expression>)
```

This is a definition of the kinds of expressions that LPL will accept and translate. Nothing is said about what they mean or how they are translated, that is for later.

To paraphrase these definitions: an <arithmetic expression> is defined as a <primary> (defined later) followed by an <expression>. An <expression> is one of three possible alternatives. An <expression> can be the terminal symbol + followed by a <primary> and another <expression>, or an <expression> can be the terminal symbol - followed by a <primary> and another <expression>, or an <expression> might be nothing at all. A <primary> is either an <identifier> (a variable name), an <integer>, or an <arithmetic expression> enclosed in matching parentheses.

What the phrases define is given by their structure and how they are combined in the phrase set. An <arithmetic expression> is a primary (a variable, number of expression in parentheses) followed by a string of <primary>'s separated by + or - signs. This provides a syntax for long sums of things like:

```
A + B + 12 - 34 - XYZ
```

The first <primary> is the <primary> in <arithmetic expression>, the following <primary>'s are part of the <expression>. The <primary> phrase recognizes the <identifiers>, <integers> and more <arithmetic expressions> enclose in parentheses. For example:

```
(A + B) - (CCC + 34)
```

LPL in BNF

With this basis for describing the syntax of programming languages we will now attempt a complete definition of LPL. Included with each phrase of the definition will be and English description of the construct that the phrase describes. Also included will be a definition of the semantics of the construct, that is, what the construct actually does when it is executed. The goal of this definition is to provide a sound framework for the Little META implementation of the LPL compiler.

BNF of LPL

1. <LPL program> ::= <statement list> END;

   An LPL program is a list of statements (to be defined) the last of which is the END statement which is terminated by a semicolon. The first statement of an LPL program is the first one to be executed. Statements are executed in order of their appearance until the END statement is reached. Flow of execution can be modified by transfer statements.

2. <statement list> ::= <statement>; <statement list> | <nothing>

   A statement list is any number of statements separated by semicolons. A statement list might also be nothing.

3. <statement> ::= <unlabelled statement> | <labelled statement>

   A statement can exist by itself, or it can have a label. Labelled statements can be the objects of transfer statements.

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

   A labelled statement has an identifier as its first symbol. This

identifier is followed by a colon. The : is followed by the body of the statement.

5. <unlabeled statement> ::=
   <assignment statement> |
   <conditional statement> |
   <transfer statement> |
   <input/output statement>

A statement is any one of the four types of statements implemented in LPL. This includes an assignment statement for setting the values of variables to other values, the conditional statement (an IF statement) for altering the flow of program execution, the transfer statement for jumping to a labelled statement (a GO TO statement) and an input/output statement for communicating with the outside world.

6. <assignment statement> ::=
   LET <identifier> := <arithmetic expression>

An assignment statement is the keyword LET followed by a variable name in the form of an identifier. This is followed by the special symbol := which means assignment. This is followed by the expression which is to be assigned to the first variable.

7. <arithmetic expression> ::= <primary> <expression>

8. <expression> ::= + <primary> <expression> |
   - <primary> <expression> | <nothing>

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

Arithmetic expressions were defined earlier.

10. <conditional statement> ::= IF <relational expression>
THEN <unlabeled statement>

A conditional statement is the keyword IF followed by a relational expression. If this expression is true, then the unlabelled statement following the keyword THEN is to be executed. If the expression is not true, then the statement which follows the IF ... THEN ... statement is executed and the unlabelled statement is skipped.

11. <relational expression> ::=
<arithmetic expression> <operator> <arithmetic expression>

Only simple relational expressions are implemented.

12. <operator> ::= = | < | >

Only three relations between arithmetic expressions are implemented. Two arithmetic expressions can be equal or one can be less than or greater than the other. The relational expression used in the conditional statement is either true or false based on the values of the arithmetic expressions computed.

13. <transfer statement> ::= GO TO <identifier>

A transfer statement is the keywords GO and TO followed by an identifier which should appear as the label on some labelled statement. The effect of executing a GO TO is to transfer program control to the labelled statement.

14. <identifier list> ::= <identifier>, <identifier list> |
<nothing>

An identifier list is a list of variable names separated by commas.

15. <input/output statement> ::= INPUT <identifier list> |
    OUTPUT <identifier list>

    The INPUT statement will accept a value from an I/O device and place that value in the variable(s) named. The OUTPUT statement will display the value of a variable (or variables) on some I/O device.

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).

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. It will use only a few features of Little META.

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

3. The interpreter will be further augmented to implement the compiler. This program will accept LPL programs and produce Z80 assembly code. It uses nearly all the features of Little META.

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.

Little META Syntax

Just as as BNF description of a programming language syntax is made up of phrases, a Little META implementation of a programming language is made up of rules which are nearly the same as BNF phrases. A Little META rule is described by the following BNF phrase:

<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.

Like BNF, 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 ('). The simplest BNF phrase in the LPL description is that of the relational operators. In BNF it was:

<operator> ::= = | < | >

In Little META this phrase is implemented in the following fashion:

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

Notice the similarity. For the most part, Little META syntax analysis will be almost the same as the BNF 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 (token) 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. This is 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 in BNF. Thus the <arithmetic expression> phrase in BNF is coded:

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

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

Rather than try to implement a Little META rule which matches <nothing> a special syntax is implemented which permits a rule to succeed when none of its alternatives do. When one or more alternatives, one of which is <nothing>, the alternatives are enclosed in brackets < ... >. Thus the <expression> phrase is implemented:

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

The <nothing> 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, or an EXPRESSION nothing at this point. Note that EXPRESSION always succeeds no matter what is at the source string at this point.

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 no answer: yes the source string is a valid LPL arithmetic expression, or no 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 problem 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 which is the list attached to the Input/Output statements, and

the list of statements separated by semicolons that forms the body of LPL programs. Little Meta provides a built in test for this kind of construct, the repetition. In BNF 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. This is the situation which most often occurs in programming languages where a single item may occur by itself but the presence of more than one requires separation by a punctuation mark.

Using repetition notation <identifier list> phrase is implemented:

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

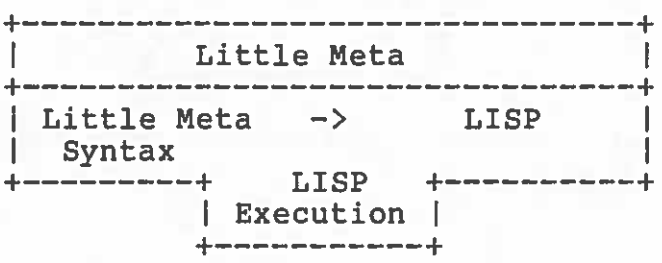
We are now in a position to code the entire LPL syntax scanner. In the text of the scanner, comments are prefixed by % and run until the end of the line. The rules of the Little Meta implemented parser will exactly parallel those of the BNF description. Each rule is annotated with the corresponding BNF 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.
    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.

CONDITIONAL-STATEMENT:      % 10.
    'IF RELATIONAL-EXPRESSION 'THEN UNLABELLED-STATEMENT ;
RELATIONAL-EXPRESSION:      % 11.
    ARITHMETIC-EXPRESSION OPERATOR ARITHMETIC-EXPRESSION ;
OPERATOR: '= / '< / '> ;    % 12.

TRANSFER-STATEMENT: 'GO 'TO ID ;    % 13.

IDENTIFIER-LIST: ID-,- ;     % 14.
INPUT-OUTPUT-STATEMENT:      % 15.
    '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 (which stands for true) if the translator is being defined, or NIL if this is a modification being made to an old one. We will discuss the interactive features of Little Meta later.

At this point the following translation is being made:



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 actual 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. Input to the system is prefixed by the * prompt character. Output from the system has no such character.

```
*(INVOKE 'LPL-SCANNER)

 ENTERING LPL-SCANNER ...

*     INPUT DVDND, DVSR;
*     LET Q := Ø;
*LOOP: IF (DVDND - DVSR) < Ø THEN GO TO DONE;
*     LET Q := Q + 1;
*     LET DVDND := DVDND - DVSR;
*     GO TO LOOP;
*DONE: OUTPUT Q, DVDND
*     END

... EXITING LPL-SCANNER
```

There certainly wasn't much output from the scanner. But this is as it should be, the program has been accepted as a valid LPL program. We have not as yet built the compilation or interpreter parts. If something was wrong with the LPL source program, then an error should be detected.

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. Special messages are 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 the rule names will be used in error messages.

The LPL Interpreter

The second program in the construction regime is an interpreter for LPL programs. The strategy is to convert LPL programs into an internal form (LISP) and then to execute them using the LISP interpreter. Since there already is a LISP interpreter there is no sense in trying to execute the LPL source code directly as many BASIC interpreters do. 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. While it is possible for BASIC programs to create other BASIC programs, their immediate execution is usually not possible. Some indirect means of loading the created program must be found.

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 forms. We now examine some of the constructors and show how they are used by placing them in pieces of the syntax scanner.

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 (given in

BNF here):

```
<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 construction of an object proceeds building the item to place on the stack, removing items from the stack, creating new ones, referencing variables, and so on. Only when the entire object has been constructed is it placed on top of the stack.

As usual we will start with the ARITHMETIC-EXPRESSION rules and proceed up. 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 3 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 represents, in LISP, 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: second arithmetic expression, 2: 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 IF statement. Note that the extra parentheses in the constructor actually become part of the created structure. The following IF statement:

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

would be translated into:

```
(COND ((LESSP (PLUS A 12) Ø) (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 in BNF 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 not (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 any 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 VARIABLES by the ASSIGNMENT-STATEMENT and the INPUT-OUTPUT-STATEMENT 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.
    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.

CONDITIONAL-STATEMENT:      % 10.
    'IF RELATIONAL-EXPRESSION 'THEN UNLABELLED-STATEMENT
    +(COND (#2 #1)) ;
RELATIONAL-EXPRESSION:     % 11.
    ARITHMETIC-EXPRESSION OPERATOR ARITHMETIC-EXPRESSION
    +(#2 #2 #1) ;
OPERATOR: '= +EQUAL / '< +LESSP / '> +GREATERP ;     % 12.

TRANSFER-STATEMENT: 'GO 'TO ID +(GO #1) ;     %13.

IDENTIFIER-LIST: ID-,- ;     % 14.
INPUT-OUTPUT-STATEMENT:      % 15.
    '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
```

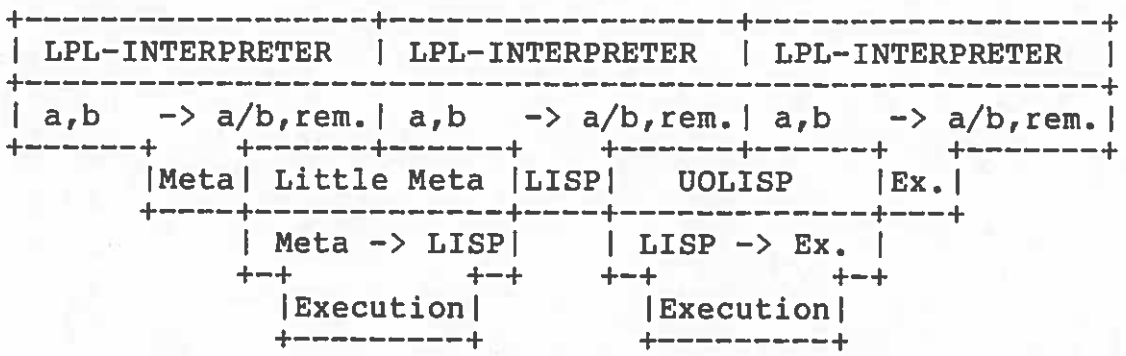
Let us now try the demonstration LPL program. We will divide the number 15 by the number 6 which should give us 2 with remainder 3 as the output.

```
*(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
*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.

If we consider this LPL-program as a translator of dividend and divisor' into quotient and remainder, the following T diagram depicts

the process of converting the program into LISP and executing it.



By turning on the UOLISP compiler we could actually cause the LPL-program to be converted from LISP into executable machine code. Our goal, however, is to implement an LPL-compiler which produces code capable of executing without the presence of the LISP interpreter. The LISP compiler generates code with many hooks into the LISP interpreter.

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 Z8Ø 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 Z8Ø instructions in the Zilog format [5]. 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.

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 $X IN VARIABLES COLLECT
      =>($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.
    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", /) ;

CONDITIONAL-STATEMENT: 'IF RELATIONAL-EXPRESSION      % 10.
    'THEN
    =>(COL:7, "JP", COL:15, #1, ",", $1, /)
    UNLABELLED-STATEMENT
    =>($1, COL:7, "EQU", COL:15, "$", /) ;
RELATIONAL-EXPRESSION: ARITHMETIC-EXPRESSION      % 11.
    =(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 ;      % 12.

TRANSFER-STATEMENT: 'GO 'TO ID      % 13.
    =>(COL:7, "JP", COL:15, #1, /) ;

IDENTIFIER-LIST: ID-,- ;      % 14.
```

```
INPUT-OUTPUT-STATEMENT: 'INPUT IDENTIFIER-LIST     % 15.
    .(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", /) ;

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      Ø
DVSR    DW      Ø
DVDND   DW      Ø
        END     GØØØ1

... 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.

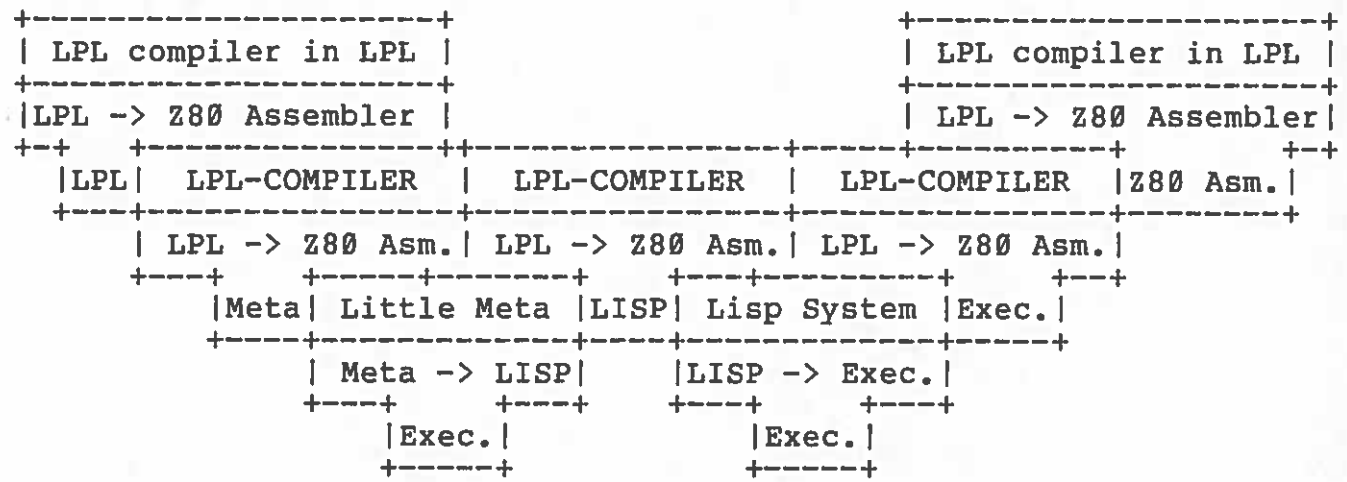
3. Many programs implemented in the language are implemented and debugged using the interpreter. During this process new features are added, removed, and old ones clarified as the need arises.

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.

The process of "bootstrapping" a compiler using this method is depicted in the following T-diagram. The final result is an LPL compiler written in Z80 assembly language.



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 [6]. 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 tool for the serious programming language enthusiast. With a small amount of practice translators can be built to perform most any task. 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 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.

3. To implement translators from very high level languages into LISP. These languages will permit concurrent execution on multiple processors, and very high level operations on data structures.

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 and "UOLISP", CIS-TR-80-18, U. of Oregon, Department of Computer and Information Science, Eugene, Oregon 97403, latest revision December 1981.

2. J. Marti, "Little Meta", CIS-TR-81-06, U. of Oregon, Department of Computer and Information Science, Eugene, Oregon 97403, latest revision December 1981.

3. Mckeeman, W. M., Horning, J. J., Wortman, D. B., "A Compiler Generator", Prentice-Hall, Inc., Englewood Cliffs, N. J., 1970.

4. Nauer, P., et al., "Revised Report on the Algorithmic Language ALGOL 60", Communications of the ACM, 3 (1960), 299.

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

6. Halstead, M. H., "A Laboratory Manual for Compiler and Operating System Implementation", American Elsevier Publishing Company, Inc., New York, 1974.