October 1981

CS-TR-81-6

LITTLE META TRANSLATOR WRITING SYSTEM

by

Jed B. Marti

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

ABSTRACT

Described is the syntax and operation of the Little META Translator Writing System, an environment for the implementation of translators, interpreters, and compilers. The system is designed to operate with the Little Big LISP system.

LITTLE META TRANSLATOR WRITING SYSTEM

The Little META translator writing system operates as program on top of the LISP interpreter. As a language i supports: BNF like syntax, recursive descent parsing schemes lexical primitives, pattern directed code generation an optimization and automatic syntax error message generation This chapter describes the operation of the system.

1.0 INTRODUCTION

The META II compiler writing system conceived by Schorre [1 was improved and implemented in LISP by Jenks [2] and calle META/LISP. The system was reimplemented by Loos [3] in LISP The system was greatly modified and enhanced by the author [4 with Martin and Cedric Griss, and Robert R. Kessler. Kessle implemented the pattern matching structures [5]. The subse system is derived from the larger one and does not implemen features relying on COMPRESS and EXPLODE.

The intended use of Little META is in languag experimentation with emphasis on features most required by th translator implementor. The goal of the system is to produce program capable of translating one language into another an possibly to assign meaning to the result by execution.

A translator from a source language to an object languag is described by a set of rules. Rules consist of tests t match strings of the source language, generators for th generation of the object language, and imperatives to affec the internal state of the translator. One rule serves as th root of the language.

The structure of a rule set resembles a BNF description o a, language. Each rule generally deals with a specifi syntactic construct consisting of one or more alternatives, each alternative being a possible form of the entity describe by the rule. Each alternative is a concatenation of the aforementioned tests, generators, and imperatives which are processed in the order of their appearance.

2.0 AN EXAMPLE

The use of Little META is typified by a recursive descen parser which analyzes simple arithmetic expressions wit addition, subtraction, multiplication, and division in thei left associative forms and exponentiation in a righ associative form. The usual operator precedence applies a well as modification by the use of nested parentheses. In BN this language is described by the grammar in Figure 1.

```
<expression> ::= <term> ;
<term> ::= <factor> | <term> + <factor> | <term> - <factor>
<factor> ::= <secondary> | <factor> * <secondary> |
             <factor> / <secondary>
<secondary> ::= <primary> | <primary> ** <secondary>
<primary> ::= <number> | ( <term> )
```

Figure 1 BNF for a simple arithmetic expression language.

The rule set which defines an interpreter for this grammar i given in Figure 2.

```
EXPRESSION:  TERM @; .(PRINT (EVAL #1)) ;
TERM: FACTOR TERMP ;
TERMP: < '+ FACTOR +(PLUS #2 #1) TERMP /
         '- FACTOR +(DIFFERENCE #2 #1) TERMP > ;
FACTOR: SECONDARY FACTORP ;
FACTORP: < '* SECONDARY +(TIMES #2 #1) FACTORP /
           '/ SECONDARY +(QUOTIENT #2 #1) FACTORP > ;
SECONDARY:  PRIMARY <'** SECONDARY +(EXPT #2 #1)> ;
PRIMARY: NUM / '( TERM ') ;
FIN
```

Figure 2. Implementation of a simple arithmetic language.

The four rules EXPRESSION, TERM, FACTOR and PRIMAR correspond to the four rules of the BNF notation. The tw rules TERMP and FACTORP are added to implement left associatio for +, -, *, and /. Little META translates all the rules int LISP functions with the corresponding names to form a recursiv descent parser for simple arithmetic expressions. T paraphrase each rule: the EXPRESSION rule parses a TER followed by the grammar terminating symbol, a semicolon. I the expression has been properly parsed, the parse tree is evaluated by LISP's EVAL function and the value printed. A TERM is a FACTOR followed by a set of FACTOR's separated by + or -. The TERMP rule forms this expression by recursing on itself. Notice that + and - are changed into PLUS and DIFFERENCE by this rule. A FACTOR is a set of SECONDARY's separated by * or /. Like TERMP, FACTORP forms the left associative form of the expression. A SECONDARY is a PRIMAR possibly followed by ** and another SECONDARY. Using this form gives the usual right associative parsing for exponentiation. The brackets <...> are a way of factoring the PRIMARY out of both alternatives. In Figure 3 some sample inputs, generated S-expressions, and their values are displayed.

INPUT	S-EXPRESSION	VALUE
12	12	12
3+4	(PLUS2 3 4)	7
2 + 3 + 4	(PLUS2 (PLUS2 2 3) 4)	9
2 + 3 - 6	(DIFFERENCE (PLUS2 2 3) 6)	-1
1 + 2 * 6	(PLUS2 1 (TIMES2 2 6))	13
(3+4)**2	(EXPT (PLUS2 3 4) 2)	49

Figure 3. Input, Generated S-expressions, and Output.

3.0 CONSTRUCTS

A rule has a name and a body describing the actions the rule takes. The rule name is separated from its body by a colon. The rule is terminated by a semicolon. Rule names should be chosen carefully to represent the rule's action as error messages in the generated translator make use of them.

A rule succeeds and returns T when one of its alternatives has all of its tests succeed. A rule fails and returns NIL when none of its alternatives succeeds

3.1 Tests

A test succeeds or fails based on syntactic or semantic information derived from the source language text. The simplest test is a quoted string which succeeds if the next token in the input string matches the given string. Quoted strings are either identifiers called keywords (LISP style identifiers), single punctuation marks, or diphthongs made of 2 punctuation marks. Non-alphabetic characters in identifiers must be prefixed by the escape character ! in both the rule set and the source language. Diphthongs do not use the escape character. Consequently they must be separated from other tokens in the META source text by at least one blank. Little META limits the length of diphthongs to 2 punctuation marks.

The rule BLOCK!-START has a single test for the diphthong consisting of two less than signs without intervening blanks. The RUNSTAT rule recognizes the keyword RUN followed by the single punctuation mark period.

```
BLOCK!-START:  '<< ;
RUNSTAT:  'RUN '.  ;
```

Note that the semicolons are separated from the rest of the rule by a single blank to assure that they do not forn diphthongs with the last character of the quoted test.

So that the last token of a source language program need not be followed by another token, the delimiter test is implemented. The rule LAST!-TOKEN succeeds on the last toker of a PASCAL program and does not require it to be followed by any other token.

```
LAST!-TOKEN:  @.  ;
```

3.1.1 Addind Diphthongs - The system automatically creates the internal tables for keywords and delimiters for a translator. Not so for diphthongs. There are several rules for the use of diphthongs. First, diphthongs cannot have more than 2 characters in them the first of which must not be alphanumeric. Second, no two diphthongs may have the same first character. Finally, diphthongs must be declared manually by the user before the execution of the translator built by Little META. This is done in the following manner:

```
(PUT '<root-rule> 'DPS
   '((a.(b .  ab)) (c.(d .  cd)) ...) )
```

Here a and b are the two characters of the diphthong ab, c and d are the two characters of the diphthong cd and so on. Diphthongs and diphthong characters are always treated as identifiers. To create two diphthongs => and **, the following is necessary:

```
(PUT 'TEST1 'DPS
   '((!=.(!> .  !=!>)) (!*.(!* .  !*!*))))
```

The grammar TEST1 may now use these diphthongs and tests for them in the usual manner.

3.2 Lexical Primitive Functions

Four primitive tests are built into the system for efficiency.

ID - Recognizes an identifier and places it on the semantic stack (discussed later). Special characters in identifiers must be prefixed by the escape character !. In BNF identifiers are recognized by the grammar:

```
<special-character> ::= !<any-character>
<lead-character> ::= <special-character> | <alphabetic>
<regular-character> ::= <lead-character> | <digit>
<last-part> ::= <regular-character> |
           <last-part><regular-character>
<id> ::= <lead-character> |
           <lead-character><last-part>
```

NUM - Recognizes an unsigned integer in the range 0 to 4095. The value is placed on the semantic stack. In BNF, numbers ar recognized by the grammar:

```
<num> ::= <digit> | <num><digit>
```

STRNG - Recognizes a delimited string of characters. String are delimited by quotation marks as in "THIS IS A STRING". Quotation marks are not permitted within the string. Th string is loaded onto the semantic stack if the rule succeeds.

ANYTOK - recognizes any token (keywords and diphthong included) and places it on the stack.

3.3 Alternatives

A rule consists of one or more alternatives separated b slashes (/). The alternatives of a rule are tried one at time until one succeeds. If all alternatives fail the rul fails. If part of an alternative succeeds then the whol alternative must succeed or an error is generated. If part o an alternative may succeed but the remainder might fail, special form of alternation must be used. In this contex alternatives are denoted by double slashes (//) implementin backup and allowing context sensitive features.

In the following example, the rule VALUE succeeds whe either an identifier, an unsigned integer, or an asteris appears in the source string.

```
VALUE:  ID / NUM / '* ;
```

Languages in which statement labels are identifiers have th problem of determining whether or not an identifier is a labe or the first token of an assignment or expression. Thi problem is solved using the backup form of alternatives in th LABELLED!-STMNT rule:

```
LABELLED!-STMNT:  ID ':  LABELLED!-STMNT // UNLABELLED!-STMNT ;
```

The rule recurses on itself until the : test fails, at whic time the UNLABELLED!-STMNT rule takes over. However, th UNLABELLED!-STMNT starts at the last ID parsed rather than a the colon.

3.4 Concatenation

An alternative is a concatenation of items. The items of a concatenation are tried one at a time until one fails, causing the alternative in which the test occurs to fail. If the first test of a concatenation succeeds, the remainder must also or an error is generated. The error will be ignored if alternation with backup is being used.

The rule PRIMARY recognizes part of a simple arithmetic expression: either an identifier, a number, or a concatenation of a left parenthesis, an EXPRESSION, and a right parenthesis.

```
PRIMARY:  ID / NUM / '( EXPRESSION ') ;
```

3.5 Grouping

If part of an alternative need not succeed for the entir alternative to succeed it is enclosed in angle brackets <...>. The <...> group always succeeds no matter what happens inside. For example, an extended BASIC identifier would be recognize by the rule BASIC!-VAR:

```
DIGIT:  '1/'2/'3/'4/'5/'7/'8/'9/'0 ;
LETTER:  'A/'B/'C/'D/'E/'F/'G/'H/'I/
          'J/'K/'L/'M/'N/'O/'P/'Q/'R/'S/
          'T/'U/'V/'W/'X/'Y/'Z ;
BASIC!-VAR:  <'$>LETTER<DIGIT> ;
```

Figure 4. Extended BASIC variable name parsing.

The leading dollar sign and trailing digit are optional Recognized are A, A0, $A, $A0, and variants thereof.

A set of items may also be grouped using parentheses. Th success or failure of the group as a whole is determined b what is inside it. The group does not always succeed like th <...> group. Compare:

```
P1:  'X ('A / 'B ) 'Y ;
P2:  'X <'A / 'B > 'Y ;
```

Rule P1 recognizes both X A Y and X B Y while rule I recognizes X Y, X A Y,and X B Y. Nesting of parentheses ar brackets is permitted to any level, though too many a illegible.

3.6 Expressions

LISP S-expressions can occur in any of three different contexts:

1. An expression can be loaded onto the semantic stack using the + operator. The + operator is discussed in the next section.

2. The value of an expression may be used in any environment when the expression is prefixed by =. When used as a test, if the expression returns NIL the test fails, anything else is considered a success.

3. The value of an expression may be computed for its effect and the result ignored by using the dot operator.

The = operator is used to "fall into" LISP from the META syntax for tests. For example:

```
INTEGER!-VARIABLE:  ID =(EQ 'INTEGER (GET ##1 'TYPE)) ;
```

succeeds only if the identifier detected has INTEGER as its TYPE on its property list.

The dot operator is used to evaluate an expression for its effect.

```
MAIN:  RULE .(PRINT ##1) .(PRINT (EVAL #1)) ;
```

In the rule MAIN, both the S-expression left on the stack by RULE and the value of the expression as computed by EVAL are printed. Even if either of these has the value NIL, MAIN will succeed as long as RULE does.

3.7 The Semantic Stack

The communication of semantic information between rules is most often accomplished by the use of the semantic stack. When a rule succeeds it generally places a value or generated expression on top of this stack. Any item on the stack may be referenced or removed provided that its position relative to the top of the stack is known. The primitive rules ID, NUM, STRNG, and ANYTOK each place their parsed token on top of the stack.

The unary operator # followed by an unsigned integer n, causes the nth item from the top of the stack to be removed and has the value of this removed item. The unary operator ## followed by an unsigned integer n has the value of the nth position of the stack without removing it. The top element of the stack is 1, the second element is 2 and so on.

The two operators may appear in any of the previousl
defined contexts as well as in patterns. Thus:

```
.(SETQ A (PLUS2 #1 #1))
```

has the effect of setting the variable A to the sum of the to
two elements of the stack.

The unary operator + has as its argument an expressio
which is placed onto the top of the stack. The expression is
template, a quoted list except where META operators appear.
These operators and their expressions are evaluated before th
completed structure is placed onto the stack. The rule NUMMP
succeeds with two numbers separated by an asterisk:

```
NUMMPY:  NUM '* NUM +(TIMES #2 #1) ;
```

If 35*2 is the input expression, the semantic stack will have
as the top element and 35 as the second element. Both thes
are removed from the stack and the expression (TIMES 35 2
replaces them. Since the order of evaluation is left to right,
the form (TIMES #1 #2) would remove the wrong item because th
#1 uncovered a new top of stack.

The - prefix within a + stack form provides a simpl
concatenation operation for constructing lists to place ont
the stack. Its effect is to strip off a layer of parenthese
during the formation of an item to place onto the stack. I
the rule COMPOUND!-STATEMENT leaves on the stack a list o
parsed LISP statements, BEGIN!-END will construct a proper PRO
form for it:

```
BEGIN!-END:  COMPOUND!-STATEMENT +(PROG NIL -#1) ;
```

The - causes the list of statements to be concatenated to th
list (PROG NIL) to create the correct result. Thus if:

```
((SETQ A 1) (PRINT A))
```

was on top of the stack, the following form would be created:

```
(PROG NIL (SETQ A 1) (PRINT A))
```

3.8 Local Variables And Generated Labels

Counters, lists, switches, and the like may be introduced i
rules as local variables by prefixing their variable names wit
a $ sign. Local variables within rules are equivalent to LIS
PROG variables in the body of that rule. Their use must alway
be prefixed with a $ sign. These variables may be used acros
recursive calls in the usual manner.

Unique symbols may be generated to serve as internal labels, variables and so on by the use of the $n construct where n is an integer. Each $n serves as a place holder for a generated variable during the execution of the rule in which it occurs. Other rules using $n constructs will have different symbols generated.

Consider the PASCAL FOR loop with only a TO clause. Generation of LISP code suitable for immediate execution requires the construction of a PROG with a loop in it:

```
FORSTAT: 'FOR ID .(SETQ $INDEX #1) ':= EXPRESSION
       'TO EXPRESSION 'DO STATEMENT
     +(PROG NIL
         (SETQ $INDEX #3)
      $1 (COND ((GREATERP $INDEX #2) (RETURN NIL)))
         #1
         (SETQ $INDEX (ADD1 $INDEX))
         (GO $1)) ;
```

Figure 5. FOR loop translation into executable LISP.

The $INDEX variable contains the index variable of the FOR loop. The $1 causes the generation of a unique label first on the loop exit test and secondly as the object of the GO. The following FOR statement:

```
FOR I:=1 TO A+B DO PRINT I
```

would be translated into:

```
(PROG NIL
      (SETQ I 1)
G001  (COND (GREATERP I (PLUS2 A B)) (RETURN NIL)))
      (PRINT I)
      (SETQ I (ADD1 I))
      (GO G001))
```

3.9 Repetitions

Lists of items can be parsed using recursive grammar constructs. META provides two constructs to simplify this commonly occuring task.

The * suffix permits zero or more repetitions of a test or group of tests. It creates a list on the top of the stack formed from any items left by these rules. A phrase suffixed by a * always succeeds.

For example, the rule IDSET below recognizes a set of identifiers enclosed in parenthesis and leaves this list on the stack.

```
IDSET:  '( ID* ') ;
```

The repeated ID test fails on the first occurrence of a right parenthesis terminating the repetition. If the source language input is (BING BANG BOOF) then upon completion of IDSET, (BING BANG BOOF) is left on top of the stack.

The second iterative construct is one which parses items separated by other items. The -x- suffix following a rule parses zero or more occurrences of the rule separated by single occurrences of x. Errors occur only if the last x separator is not followed by another occurrence of the rule. Several examples follow:

```
IDLIST:  '( IDLIST-,- ') ;
```

IDLIST recognizes a list of identifiers separated by commas and parses it into a list of identifiers which it leaves on top of the semantic stack

```
COMPOUND!-STATEMENT:  'BEGIN STATEMENT-;- 'END ;
```

COMPOUND!-STATEMENT parses a BEGIN ... END block in most block structured programming languages and leaves a list of the parsed statements on top of the stack.

```
FUNCTION!-CALL:  ID '( EXPRESSION-,- ') +(#2 -#1) ;
```

FUNCTION!-CALL parses a function call and creates the prefix LISP form. If no arguments are present NIL is left on top of the stack and the correct prefix form is generated.

3.10 Recursion

Rules may be self recursive or indirectly recursive to any level. The WFF rule set parses well formed formulas into prefix form for LISP evaluation or as the front end for a theorem prover. Note that AND, OR, and IMPLIES are parsed in their left associative form.

```
WFF: TERM WFFP;
WFFP: <'IMPLIES TERM +(IMPLIES #2 #1) WFFP>;
TERM: SECONDARY TERMP;
TERMP: <'OR SECONDARY +(OR #2 #1) TERMP>;
SECONDARY: PRIMARY SECONDARYP;
SECONDARYP: <'AND PRIMARY +(AND #2 #1) SECONDARYP> ;
PRIMARY: 'TRUE +T | 'FALSE +NIL | '( WFF ') |
   ID <'( WFF[-,') +(#2 -#1)> ;
```

Figure 6. Predicate Calculus expression parsing.

The WFF and WFFP rules are the common way of parsing expressions into their left associative forms. This is echoed in TERM, TERMP and SECONDARY, SECONDARYP. The PRIMARY rule parses the constants TRUE and FALSE, WFF's in parentheses, and identifiers as either variables, or function names with arguments.

4.0 THE SYMBOL TABLE.

To aid the user in storing attributes of identifiers a number of functions beyond LISP's simple GET and PUT are implemented. These functions aid the storing of information in block structured language variable declarations. The information so stored is divided into single _tables_ which in turn contain the unique identifiers of a block. Associated with each identifier are _flags_ and _attribute_ - _value_ pairs. Each table has a name and is created before it is needed by the MKTABLE function.

(MKTABLE <table-name>)

The function creates an empty symbol table whose name is <table-name>. If the table is no longer needed its space can be released to the system by the CLEAR function.

(CLEAR <table-name>)

To enter information in a table the ENTER function is used:

(ENTER <identifier-list> <attribute/flag> <table-name>)

Here <identifier-list> is a list of one or more unique identifiers. Identifiers which are already in this particular table have their attributes modified, those which are not in the table are added with a single attribute or flag. If the <attribute/flag> quantity is an identifier it is added as a flag. If it is a dotted-pair, the CAR of the pair becomes the attribute and the CDR becomes its value. The CAR part must be an identifier, the CDR can be anything.

The code segment in Figure 7 parses simple PASCAL variable declarations and creates a symbol table called GLOBAL which will be accessible at all block levels.

```
VARDEC: 'VAR .(MKTABLE 'GLOBAL)
        VARS-;- }

VARS: ID-,- ':
        <'ARRAY '[ NUM '. '. NUM '] 'OF
         .(ENTER ##3 (CONS 'ARRAY (CONS #2 #1)) 'GLOBAL)>
        SIMPLE|-TYPE
         .(ENTER #2 #1 'GLOBAL) }

SIMPLE|-TYPE: 'INTEGER +INTEGER / 'CHAR +CHARACTER /
        'BOOLEAN +BOOLEAN;
```

Figure 7. PASCAL Global Variable Declaration Parsing.

The first rule creates a table named GLOBAL when the VA declaration is encountered. It then proceeds to parse all th variable declarations separated by semicolons. Each variabl declaration (VARS) is a list of identifiers separated by comma and followed by a colon. This list of identifiers might be a array in which case each identifier has the ARRAY attribut which has as a value the lower and upper numeric bounds of th array(s). The type of the variables can be either INTEGER, CHAR, or BOOLEAN. The type of the variable is entered as flag.

To display the contents of any table, the DUMP functio can be used.

```
(DUMP <table-name>)
```

Each symbol in the table is listed once followed by its flag and attributes indented two spaces. The symbols are not liste in any particular order.

With the variable declarations in Figure 8, the GLOBA symbol table is then displayed.

```
VAR I, J: INTEGER;
    A: ARRAY[0..10] OF CHAR;
    INTVAR: ARRAY[5..6] OF BOOLEAN;

I
  INTEGER
J
  INTEGER
A
  (ARRAY 0 . 10)
  CHARACTER
INTVAR
  (ARRAY 5 . 6)
  INTEGER
```

Figure 8. Parsing of some declarations and DUMP output.

Access to attributes and flags in a symbol table is restricte to two functions. The ACCESS function retrieves the value o an attribute.

(ACCESS <identifier> <attribute> <table-name>)

ACCESS will return the value associated with the attribute o the identifier if there is one and NIL otherwise.

The corresponding function for flags is ISIT which return T or NIL depending on the presence of a flag associated with a identifier.

(ISIT <identifier> <flag> <table-name>)

The code section in Figure 9 parses simple arithmeti expressions in the usual form. It also checks that th identifier names used in the expression have been declared a integers and signals an error if not.

```
IEXP: ITERM IEXPP;
IEXPP: <'+ ITERM +(PLUS2 #2 #1) IEXPP>;
ITERM: IPRIM <'* ITERM +(TIMES2 #2 #1)>;
IPRIM: ID
   (=(ISIT ##1 'INTEGER 'GLOBAL)
       <=(ACCESS ##1 'ARRAY 'GLOBAL)
         '[ IEXP '] +(GETV #2 #1)>
     / .(ERROR 0 "Not an integer variable"))
   / NUM
   / '( IEXP ') ;
```

Figure 9. Integer Expression Parsing with Error Detection.

5.0 THE PATTERN MATCHER

An alternative to code generation during parsing is to use the constructed parse tree to control a pattern directed code generator. This was first implemented in the TREE-META system [6]. The version used in META was implemented by Martin Griss and interfaced with the system by Robert Kessler [5]. The inclusion of some pattern matching primitives for type detection and an indeterminate length match have been added to aid mixed mode expression analysis and peephole optimization.

5.1 Pattern Sequences

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

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

To invoke a match against a pattern sequence the pattern name with an S-expression as its argument is used:

```
=(GCODE '(PLUS2 1 3))
```

would invoke the pattern GCODE with (PLUS2 1 3) as the S-expression to match.

5.2 Pattern Primitives

A pattern is a template against which the actual parameter of the pattern sequence is matched. If a match against a pattern succeeds the corresponding action is taken. 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 (NOW WE ARE 6) and no other. The atomic entities which may occur in patterns are identifiers, strings, and integers.

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 and has as its last element any LISP S-expression. Thus (NOW WE ARE 1), (NOW WE ARE GONE), (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. pattern is matched.

To permit alternatives within a pattern the followin construct is implemented:

```
&(test[0] / test[1] / ...  / test[n])m
```

This pattern construct succeeds if any one of the subpattern (tests) succeeds. The portion of the source expression bein matched is available as &m in the action part of the rule. I a test is an atom, it must exactly match the source expressio atom. The matched atom will be available in &m on the actio side. If the test is an identifier preceeded by an = sign then the function named by the identifier will be applied t the s-expression in the source expression. If this functio returns NIL, the test fails, any other value and the test wil succeed. Thus the following pattern:

```
(PLUS &(=IDP / =NUMBERP)1 &2) ->
```

will match lists in which the second element is an identifie or a number. The third element can be any s-expression, th first must be the identifier PLUS. Thus (PLUS 1 2), (PLUS NOW (PLUS "XXX" Y)), and (PLUS A B) are matched, bu (PLUS "BAD" 12) is not because the second element of the lis is not an identifier or a number.

5.3 The Action Side

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

1. The stack reference and access functions # and ##.
2. Quoted expressions as in '(QUOTED EXPRESSION).
3. Expressions prefixed with = as in =(EVAL '(THIS EXPRESSION)).
4. $ prefix local variables and generated symbols.
5. & pattern pieces.
6. atoms.
7. Combinations of the above in expressions.

5.4 An Example.

The pattern set in Figure 10 does reduction of constants i LISP expressions with the arithmetic operations of addition an multiplication.

```
REDUCE =
&(=NUMBERP)1 -> +&1 T,
&(=IDP)1 -> +&1 T,
(PLUS &(=NUMBERP)1 &(=NUMBERP)2) -> +=(PLUS &1 &2) T,
(PLUS &1 0) -> =(REDUCE &1) T,
(PLUS 0 &1) -> =(REDUCE &1) T,
(TIMES &(=NUMBERP)1 &(=NUMBERP)2) -> +=(TIMES &1 &2) T,
(TIMES &1 0) -> +0 T,
(TIMES 0 &1) -> +0 T,
(TIMES &1 1) -> =(REDUCE &1) T,
(TIMES 1 &1) -> =(REDUCE &1) T,
(&1 &2 &3) ->
     =(REDUCE &2) =(REDUCE &3)
     +(&1 #2 #1) T;
```

Figure 10. Constant Reduction by Pattern Matching.

Notice the order of the patterns. The most specific cases ar dealt with first. In this case single numbers and identifier are returned on the stack without evaluation. When the sum o two numbers is detected, it is replaced by the value of the su of two numbers. If there is anything added to 0, the valu loaded on the stack is the reduction of non-zero ite reflecting the identity property of zero. The same is true fo multiplication except that 1 is the multiplicative identity an 0 as either argument causes 0 to be returned. If none of th patterns matches, the last pattern will match everything. I this case the arguments of either PLUS or TIMES are recursivel reduced and the result loaded on the stack.

List of References

1. Schorre, D. V., "META II: A Syntax Oriented Compile Writing System", Proceedings ACM 19th National Conference 1964, p. D1.3.

2. Jenks, R. D., "META/LISP: An Interactive Translato Writing System", IBM Corporation, Thomas J. Watso Research Center, Yorktown Heights, New York, 1971.

3. Loos, R., private communication.

4. Marti, J. B., "The META/REDUCE Translator Writing System" SIGPLAN Notices, Vol. 13, No. 10, October 1978, pp 42-49.

5. Kessler, R. R., "The PMETA System", Utah Symboli Computation Group, Operating Note No. xx.

6. Carr, C. S., Luther, D. A., Sheridan, E., "The TREE-MET Compiler-Compiler System", RADC-TR-69-83, University o Utah, March 1969.

Index