Two-level Grammar as a Functional Programming Language

B. EDUPUGANTY* AND B. R. BRYANT

Department of Computer and Information Sciences, The University of Alabama at Birmingham, Birmingham, Alabama, U.S.A. 35294

A two-level grammar consists of two context-free grammars interacting in a manner such that their combined computing power is equivalent to that of a Turing Machine. We show that two-level grammar can be used as a functional programming language where the two-level grammar notation is the syntax of the language. Two-level grammar programs as represented are actually high-level descriptions of recursive functions, with one level of the grammar specifying the function domains and the second level defining the function evaluation rules. A general algorithm to interpret two-level grammars is given to make two-level grammar programs executable. The primary advantages of two-level grammar over other functional programming languages are (1) the capability for data and procedural abstraction provided by the different levels of the grammar and (2) the structured form of natural language which is used as the two-level grammar syntax, making two-level grammar programs very readable.

Received July 1986, revised September 1987

1. INTRODUCTION

In this paper we show that two-level grammar (also called TLG or W-Grammar) can be used as a functional programming language where the TLG notation is the syntax of the language. Two-level grammar was introduced by van Wijngaarden and are used to define the context-free and context-sensitive syntax of ALGOL 68. Uzaglis showed that two-level grammar can also be used to define the interpretive or operational dynamic semantics of programming languages. Additional studies on two-level grammar for operational semantics may be found in Cleveland, Marcotty, and Pagan. Using TLG as a programming language is a generalisation of the fact that an operational language definition is a two-level grammar 'program' to interpret that language.

The concept of using the two-level grammar notation as a non-procedural language was first proposed by van Wijngaarden. Our paper focuses upon yet another approach to refining this concept into an elegant functional programming language, which is executable using the interpretation algorithm presented in this paper. We find that two-level grammar programs are actually high-level descriptions of recursive functions and show how a TLG program can be derived from a recursive function definition of a problem. Programs written in two-level grammar notation are readable due to their natural language vocabulary, which may be used freely in writing two-level grammars. The close correspondence between the two-level grammar program and the function definition of the problem further enhances readability.

Another advantage the two-level grammar provides is the data-typing capability achievable by the metalevel of the grammar. An example is given to show how a two-level grammar programs can be developed using top-down and stepwise refinement techniques, ideas familiar to proponents of software engineering and structured programming. We claim that the primary advantages of TLG programs are readability, implementability, referential transparency, and the capability for data and procedure abstraction provided by the two levels of the grammar.

* Present address: Software Leverage, Inc., 485 Massachusetts Avenue, Arlington, Massachusetts, U.S.A. 02174.

In Section 2 we present TLG notation and some important formal properties. Section 3 introduces TLG programming, restrictions imposed on the two-level grammar programs, and our algorithm to interpret TLG specifications. In this section, examples of TLG programs are given and their interpretation is described. Section 4 illustrates the top-down development of a TLG program. In Section 5 the TLG programs are shown to compare favourably with equivalent Lisp and Prolog code to solve the same problem. We conclude with a summary of results and our extended research goals.

2. TWO-LEVEL GRAMMAR NOTATION

A two-level grammar consists of two finite grammars: the rules of the first level grammar are called metaproduction rules (or metarules) and the rules of the second level are called hyperrules. The first level grammar generates the non-terminal elements of the second level grammar. That is, the hyperrules are actually rule schemata with place holders called metavariabes or metanotions. From these two finite sets of rules a third, possibly infinite, set of production rules is derived.

A protonotion is a (possibly empty) sequence of lower case letters (e.g. head and tail). A metanotion is a non-empty sequence of upper case letters, with embedded underscores for readability (e.g. LINKED_LIST). A hypernotion is a (possibly empty) sequence of metanotions and/or protonotions (e.g. head of LINKED_LIST). The metanotions are defined by the metavarules, wherein the metavariabes can be considered as the non-terminals and protonotions as the terminal symbols. The format of the metaproduction rules is:

\[ \text{METANOTION}_1, \ldots, \text{METANOTION}_n, \text{HYPERNOTION}, \ldots, \text{HYPERNOTION}, \{m, n \geq 1\} \]

For convenience we allow metavariabes with the same right-hand side to be declared in one rule, with the metavariabes on the left separated by commas (as in high-level language variable declarations). Ordinarily, the metavarules take the form of a context-free grammar. If a metavarule defines only a regular set, then the hyperrules

Downloaded from https://academic.oup.com/comjnl/article-abstract/32/1/36/341935 on 29 April 2018
on the right-hand side may be written as regular expressions.

The format of the hyperrules is:

hypernotation: hyperalternative\(,\) hyperalternative\(n|n \geq 1|\) where each hyperalternative is a list of hypertonotions separated by commas. It should be noted that the \`\`;\`\` in the hyperrule serves the same purpose as a \`\`\`\` in a Backus–Naur Form (BNF) grammar rule, and the \`\`\`\` serves the same purpose as a space between grammar symbols in the right-hand side of a BNF grammar rule. Each hypertonotation in a hyperalternative corresponds to a branch of the derivation tree. A hyperrule can possibly generate one or more production rules. Deriving a production rule from a hyperrule consists of applying the following:

1. Each metatonotation found only once in the hyperrule is replaced with a terminal string of productions derived by the metaproduction of that metatonotation.
2. When the same metatonotation occurs two or more times in the hyperrule, each occurrence of that metatonotation is replaced with the same terminal string. This is called the Uniform Replacement Rule (URR) or Consistent Substitution.

To avoid consistent substitution, metatonnotations may be subscripted. Subscripted metatonnotations followed by a number represent the same root notion, but allow more flexibility with no increase in grammatical power (e.g. in the hypertonotation where \(\text{NUMBER1 NUMBER2 equals NUMBER1}\), the two instances of \(\text{NUMBER1}\) should be the same, and the two instances of \(\text{NUMBER2}\) should be the same, but \(\text{NUMBER1}\) need not be the same as \(\text{NUMBER2}\).

Some hyperrules may be used as predicates to act as conditions which must be satisfied. Predicates were introduced to enforce context-conditions required to describe the context-sensitive syntax of programming languages. A predicate generally begins with the word \(\text{where or is}\), its terminal derivation is empty (signifying truth of the predicate).

It should be pointed out here that a TLG in which the hyperrules are ambiguous or have more than one hyperalternative is nondeterministic; whereas if each hyperrule has only one hyperalternative and there are no ambiguous hyperrules the TLG is deterministic. In general, a TLG may be written either deterministically or non-deterministically.

### 2.1. Formal Properties

The two-level grammar notation, presented informally in the previous section is now formalised, and the formal properties of two-level grammar relevant to our paper are stated. Additional studies on the formal properties of W-grammars are found in Baker, Deussen, Greibach, Kupka, Sintzoff and Wegner.

A two-level grammar can be formally defined as a 6-tuple \((M, V, T, R_M, R_V, S)\), where

- \(M\) is a finite set of metatonotions;
- \(V\) (protonotions) is a finite set of syntactic variables such that \(M \cap V = \emptyset\);
- \(T\) the terminal vocabulary, is a finite subset of \(V^*\);
- \(R_M\) is a finite set of metarules \(X \rightarrow Y\), where \(X \in M\), \(Y \in (M \cup V)^*\) or \(Y\) is a regular expression, and for all \(W \in M, (M, V, R_M, W)\) is a collection of context-free grammar (CFG) and regular expression rules;
- \(R_V\) is a finite set of hyperrules of the form \(H_0 \rightarrow H_1, H_2, \ldots, H_m\), where \(m \geq 1\) and \(H_i \in (M \cup V)^*\), \(H_i \in (M \cup V)^*\) for \(i \geq 1\), \(H_i\) is a hypertonotation; \(S \in (M \cup V)^*\), the start notion.

This definition of the start notion differs from that in the original definition of a TLG, and will be used as a mechanism to define input (function arguments) to a TLG program. A significant result is due to Sintzoff, who proved that for every Type 0 phase-structure grammar there is an equivalent two-level grammar (i.e. Turing equivalence). Turing equivalence of two-level grammar assures that every two-level grammar can be implemented and that for every algorithm there exists a TLG to define that algorithm. Using an infinite number of derived production rules, two-level grammars can generate the context-sensitive and recursively enumerable languages. If only a finite number of production rules can be derived, then the two-level grammar has no more descriptive power than a context-free grammar.

#### 2.2. Applications of Two-Level Grammars

In this section a brief survey of the relevant literature is presented, and a comparison between our work and previous work is given. Kupka showed that TLG can represent computable functions and generalised van Wijngaarden’s non-procedural problem solving method to a formal model for information transformation. He introduced the idea of state-oriented programming as the main application of the model. He proposed that the state-oriented technique be used in program development (non-procedural and non-executable) which can then be translated to an algorithmic language. Van Wijngaarden showed the feasibility of using TLG notation for algorithms without an intervening programming language.

Hesse showed how TLG can be used to describe logical calculi. Using a form of TLG called PW-grammar (predicate W-grammar) to describe first order predicate logic, he gave an axiomatic definition of Hoare’s simple structured language (which we call HSSL) and showed how programs in HSSL can be verified. It is our opinion that his paper also forms the basis for using TLG notation for logic programming. Bryant, Edupuganty and Hull developed a framework for an automatic program verification system which showed how program verification using TLG could be accomplished within the framework of a language definition.

Maluszyński introduced the concept of a logic programming language based on TLG. A notion of the relation specified by a TLG (non-algorithmic specification) was also introduced and the computability of such relations was discussed. He introduced the transparency condition which makes every hypertonotation of a transparent TLG into a representation of a tree structure. For transparent TLG’s the grammatical unification problem is decidable and reduces to the usual term unification. (Grammatical Unification can be described as the problem of determining whether two hypertonotions)
$h_1$ and $h_2$ are equal under some homomorphic replacement of metamotions.) A general algorithm is given for computing the relation specified by a transparent TLG using usual term unification and context-free parsing. Maluszynski also showed that the transparent TLG is a generalisation of Horn clauses and defines a version of Prolog called Metaprolog\cite{29} based on the notion of TLG. Turner\cite{30} described a system called WLOG (W-Grammar Logic) for the execution of hyperrules on a data flow architecture, and compared the advantages of using TLG notation for Prolog with those of existing implementations of Prolog.

Other applications of TLG include a generative method for the design of data base applications,\cite{31} the specification of natural language syntax,\cite{32} executable denotational definitions of programming languages,\cite{33} and the applicability of TLG notation for data-flow and pipelined systems.\cite{34}

3. TWO-LEVEL GRAMMAR PROGRAMMING

In recent years there has been considerable interest in functional programming.\cite{35,36,37} It is our view that one of the important features of a functional programming language is that there should be an easy transformation from a function definition of a problem to the program. Unfortunately, some of the popular functional programming languages have not been able to incorporate this feature. We will show that every two-level grammar 'program' is actually a recursive function definition using the highly stylised metalanguage. Two-level grammar programming is therefore a form of functional programming. The methods developed in the next section illustrate the elegance of two-level grammar as a functional programming language.

We will not present a formal inductive definition for recursive functions, but will instead present an informal definition. (For a more formal definition refer to Manna.\cite{38}) A recursive function definition consists of a set of rules that for various arguments specify the function in terms of variables, string constants, the function itself, well defined primitive functions, conditionals, and other recursive functions or an expression built from the above using functional composition. Intuitively, a recursive function definition can be envisaged as:

$$f(x_1, x_2, \ldots, x_n) = y$$

where the $x_i$'s are string constants or variables and $y$ is a string constant or another recursive function (including the function being defined). We discuss the restriction that the arguments to a function must be atomic values after we present the interpretation of TLG programs. It should be noted that this definition easily extends to arithmetic functions since any integer may be represented by a unary alphabet.

A partial recursive function is a function that is defined only on part of its domain. That is, the function is undefined for some values of the arguments. The class of partial recursive functions is exactly the class of Turing computable functions and total recursive functions are the ones for which a Turing machine exists that halts on all inputs. In terms of the above definition, if the function $f$ is defined for all $x_1, x_2, \ldots, x_n$ then the function $f$ is a total recursive function; otherwise $f$ is a partial recursive function. We don't distinguish between total and partial recursive function definitions because such a distinction does not affect the way a recursive function definition is written.

In a two-level grammar program the start rule can be considered as a high level specification of the problem, with the input to the program represented by metamotions ranging over the appropriate domains. The output of the program is represented by an additional metamotion, which is said to be synthesised. In terms of the function definition given above, the start rule represents a function of arity $n$ ($n \geq 0$), and the output ($y$ above) is returned by a metamotion. The metalanguage of the grammar specifies the names of the arguments to the function (each hypernotion can be considered as a function) and also specifies their domains. We claim that this capability of TLG to define the domains of functions represents a significant advantage over other functional programming languages since it allows increased data abstraction and typing. We illustrate these advantageous features of TLG in Sections 3.3 and 3.4.

3.1. Restrictions on Two-Level Grammars

The mathematical paradigm on which TLG programming is based is Turing equivalence. We now introduce the restrictions on TLGs which make their interpretation deterministic.

C1. The metarules must be unambiguous and context-free.

C2. The hyperrules must be context-free and uniquely assignable.

These restrictions also preserve Turing equivalence. It is well known that ambiguity of context-free grammars is an undecidable problem, but one way to guarantee unambiguity is to make the metarules LR(1). What we mean by 'deterministic' in C2 is, firstly, no hyperrule has more than one alternative on the right-hand side and, secondly, no two hyperrules have the same left-hand side. Informally, unique assignability ensures that given a 'string' and a hypertonion on the left hand side, the pattern matching (used in interpretation) will be able to uniquely determine the substring derivable from each metatnation in the hypertonion. Unique assignability is undecidable for context-free metarules but judicious writing of the hyperrules can satisfy this condition. (For a more formal discussion the interested reader is referred to Wegner.\cite{39}) C1 and C2 are essential restrictions, since the interpretation algorithm does not backtrack. Since the restrictions impose certain conditions on the context-free grammars but do not affect their grammatical power, it follows that every recursively enumerable set can be generated by a TLG satisfying restrictions C1 and C2.

3.2. Two-Level Grammar Interpreter

We have developed a method of interpreting two-level grammar specifications (satisfying restrictions C1 and C2 given above) which makes TLG programs executable. Thus we have the capability of formally specifying an algorithm which is directly executable. In our method, the interpretation or execution of a two-level grammar program proceeds by expanding the start notion and
applying the hyperrules to all of the branches of the derivation tree until all the leaves of the tree derive the empty string. This signifies the completion of the interpretation. The derivation tree of the two-level grammar represents the control flow of the program execution. 'Execution' consists of matching each hyperton as a sentential form, represented by a node in the tree, with the left-hand side of a hyperrule, and then spawning new branches from the sequence of hyperton as on the right-hand side of the hyperrule. The restrictions imposed on writing TLGs ensures that matching (similar to the problem of determining which rule to apply in a term rewriting system) can be accomplished using only context-free parsing. In a determinist TLG, there will be at most one hyperrule whose left-hand side matches any hyperton in a sentential form. If there are more than one, the interpretation must be non-deterministic. It should be noted that if the right-hand side of a hyperrule is the empty string, then we reach a leaf node of the tree. Execution begins at the root node (start hyperton), and proceeds to each branch of the derivation tree recursively from left to right (an in-order traversal). For functional TLG programs the order of execution can be defined as call-by-value (also called a leftmost-innermost computation rule in Manna).

During the execution of the two-level grammar derivation tree there may be some metanotions whose instantiations are unknown during the expansion. Such metanotions are 'synthesised' (instantiated) during the expansion process. Synthesis occurs when the metaton must have a particular value for the left-hand side of a hyperrule to match the hyperton being executed. Synthesised metanotions may be passed down the derivation tree any number of levels before they are instantiated. When a metaton is synthesised by some branch of the derivation tree, the value is propagated up the tree, thus instantiating all occurrences of that metaton at a particular level in the tree. This is similar to the concept of synthesised attributes in attribute grammars. Predicates do not contain synthesised metanotions, as they are used to enforce conditions.

As mentioned earlier, the execution (or interpretation) proceeds until all the leaf nodes derive the null string. If, during an expansion step, the match fails (signifying that there is no hyperrule to expand), then we say that a 'blind-alley' has been reached. Blind alleys imply that the program contains an error. An error is also indicated if a predicate does not derive the null string (signifying that the predicate holds).

The execution of a hyperton can be considered as a function call with the synthesised metanotions used to return values evaluated by the function. The restrictions that the arguments to a function must be atomic or base values (i.e. non-functional values) is primarily due to the way the interpretation algorithm works. The above restriction is not a severe limitation, since functional composition can be decomposed into a sequence of function applications.

Having described how two-level grammar definitions are interpreted, we will now introduce the interpretation algorithm.

 procedure expand (hyperton)
1. find the hyperrule to apply which has the hyperton as its left-hand side. This rule will be of the form hyperton: hyperton-1,...,hyperton-n.
2. expand the derivation tree with hyperton as the root and the branches being the n hyperrules on the right-hand side of the rule found in step 1.
3. for i = 2,...,n do
   expand (hyperton-i)
4. return the value of any synthesised metanotions in the argument to the procedure.

It should be pointed out here that the interpretation algorithm presented is similar to the procedural interpretation of Prolog programs.

3.3 Two-Level Grammar for the Palindrome Problem

We now consider the problem of determining whether a given string is a palindrome. A string of length zero or one is a palindrome. A string of length greater than one is a palindrome if the first and last characters are the same and the rest of the string (between the first and last characters) is a palindrome. The recursive function definition and the TLG program for the palindrome problem are given in Fig. 1, where 'x' is a single symbol over some alphabet \( \Sigma \). 'e' is the empty string, and 'a' is a string of length greater than or equal to zero (i.e. \( x \in \Sigma^* \)). The value 'true' has been used to indicate that the given string is a palindrome.

Note that the input alphabet in the example is \( \Sigma = \{a, b, \ldots, z\} \). The metaton LETTER specifies the domain of \( \Sigma \), and the input to the program is represented by the metaton STRING. Since STRING is defined in terms of LETTER, the domain of strings representable by the metaton STRING is thus specified. The start notion is STRING a palindrome. The first hyperrule takes care of the case where the given string is 'empty' or the null string. The second hyperrule takes care of the case where the length of the given string is one, and the third rule is used when the length of the string is greater than or equal to two (i.e. the general case). The way the algorithm works is that hyperrule three is applied recursively until one of the other rules is applicable, at which time the algorithm terminates. It should be pointed out that the TLG program shown above is deterministic since none of the hyperrules have

Recursive function definition

\[
\begin{align*}
f(e) &= true \\
f(x) &= true \\
f(xxx) &= f(x)
\end{align*}
\]

Two-level grammar program

Let
\[
\text{LETTER}::=a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z.
\]

\[
\text{STRING}::=(\text{LETTER}).
\]

\[
\text{EMPT Y}::=.
\]

Hyperrules

1. is string EMPT Y a palindrome: EMPT Y.
2. is string LETTER a palindrome: EMPT Y.
3. is string LETTER STRING LETTER a palindrome:
   is string STRING a palindrome.

Figure 1.
more than one hyperalternative and are all uniquely assignable. We would like to also emphasise the correspondence of the function definition to the hyperrules shown above and the readability of the TLG program.

Our method of interpreting TLG programs can be described by following the execution trace for an instance of the application of the above program. Given below is the trace of the control flow of the program, given the string ‘xyxyx’. The execution trace can be considered as an interpretation tree wherein the branches are the hypernotations spawned by the application of the hyperrules.

The application of the start notion yields branch ‘I’ (the root node), where the value of the metanotion STRING is ‘xyxyx’. The root notion is then matched against the left-hand sides of other hyperrules. For example, matching the hypernotation is string xxyyx a palindrome with the left-hand side of rule 3 is string LETTER STRING LETTER a palindrome, it is evident that the metanotion LETTER matches x and STRING matches ‘yy’. It can be verified that rules one and two would not match (due to the length of the string). This matching of hyperrule left-hand sides we call an ‘expansion’. The expansion step yields the sub-tree shown below.

1. ● is string xxyyx a palindrome
2. ○ is string yyx a palindrome

The execution continues until all the leaf nodes derive the null string, or until no rules are applicable (i.e. a ‘blind alley’ has been reached signifying that the input string is not a palindrome). Expanding branch 2 by applying hyperrule 3 and then applying hyperrule 1 to the resulting node yields the null leaf node. Execution thus terminates, signifying that the given string ‘xxyyx’ is a palindrome. The sub-tree with branch 2 as root is shown below:

2. ● is string yyx a palindrome
3. ○ is string ε a palindrome
4. ●

The complete execution control tree for this example is the following:

is string xxyyx a palindrome
is string yyx a palindrome
is string ε a palindrome

3.4. Two-Level Grammar for Finding the Maximum Number in a List

Consider the problem of writing a two-level grammar program for finding the maximum of a given list of numbers. The algorithm used is the familiar linear search algorithm. A recursive function definition and the formal definition using two-level grammar notation (i.e. a TLG program) for the linear search program is given in Fig. 2.

The TLG program for linear search works exactly in the same fashion as the recursive function definition. The metamotions FIRST_NUMBER, CURRENT_MAX and MAX are of type integer. The metamotion REST_OF_NUMBERS represents a list of numbers which may be empty. The use of these metamotions should be obvious from their names.

Recursive function definition

1. \( f(\text{list}) = g(\text{tail}(\text{list}), \text{head}(\text{list})) \)
2. \( g(\text{nil}, \text{max}) = \text{max} \)
\( g(\text{list}, \text{max}) = \left\{ \begin{array}{ll}
\text{if} & \text{max} < \text{head(list)} \\
\text{then} & g(\text{tail(list)}, \text{head(list)}) \\
\text{else} & g(\text{tail(list)}, \text{max})
\end{array} \right. \)

Two-level grammar program

Metarules

FIRST_NUMBER, CURRENT_MAX, MAX::INTEGER.
NUMBERS:::(INTEGER)+,
REST_OF_NUMBERS:::(INTEGER)*.

Hyperrules

1. find maximum of NUMBERS:
given a list of numbers NUMBERS
 find the maximum number MAX, the maximum number is MAX.
2. given a list of numbers
 FIRST_NUMBER REST_OF_NUMBERS
 find the maximum number MAX:
given a list of numbers REST_OF_NUMBERS
 and current maximum FIRST_NUMBER
 find the maximum number MAX.
3. given a list of numbers EMPTY
 and current maximum MAX
 find the maximum number MAX: EMPTY.
4. given a list of numbers
 FIRST_NUMBER REST_OF_NUMBERS and current maximum CURRENT_MAX
 find the maximum number MAX:
 if CURRENT_MAX is less than
 FIRST_NUMBER then
given a list of numbers
 REST_OF_NUMBERS and current maximum FIRST_NUMBER
 find the maximum number MAX
 otherwise
given a list of numbers
 REST_OF_NUMBERS and current maximum CURRENT_MAX
 find the maximum number MAX.

Figure 2.

The first hyperrule is the start rule and helps to decompose the problem into finding the maximum number and defining the output. The second hyperrule corresponds to step 1 of the algorithm and recursive function \( f \) above. Hyperrules three and four correspond with step 2 of the algorithm and recursive function \( g \). Hyperrule four is applied recursively until the list of numbers has been traversed. Hyperrule three is the termination condition (i.e. empty list). The conditional statement of hyperrule four works in the usual manner, and a separate rule has not been shown for this. In hyperrule one, the value of metanotion MAX is initially unknown and remains so until hyperrule three is applied, at which time its value is instantiated. The metanotion MAX is called a synthesised metanotion; after its value is instantiated, this value is propagated up the tree. That is, the value of the metanotion MAX is known after the execution of the sub-tree corresponding to the first hypernotation of the start hyperrule.

We now trace the application of the TLG program for linear search given the numbers 4, 16 and 8.
0  find maximum of 4168
1a  given a list of numbers 4168
   find the maximum number MAX
1b  the maximum number is MAX

The expansion of branch 1a by the application of hyperrule two is:
1a  given a list of numbers 4168
    find the maximum number MAX
2  given a list of numbers 168 and
    current maximum 4 find the
    maximum number MAX

Expansion of branch 2 by rule four is:
2  given a list of numbers 168 and
    current maximum 4 find the maximum
    number MAX
3  if 4 is less than 16 then
    given a list of numbers 8 and
    current maximum 16 find the
    maximum number MAX
otherwise
    given a list of numbers 8 and
    current maximum 4 find the
    maximum number MAX

This is an ‘if’ statement and since the condition is true
the statement associated with the ‘then’ will be executed
and this is shown by branch 4 below.

4  given a list of numbers 8 and
    current maximum 16 find the
    maximum number MAX
5  if 16 is less than 8 then
    given a list of numbers and
    current maximum 8 find the
    maximum number MAX
otherwise
    given a list of numbers and
    current maximum 16 find the
    maximum number MAX

Since the condition is false, the statement associated with
‘otherwise’ is executed yielding branch 6, which is
expanded using hyperrule three.

6  given a list of numbers and
    current maximum 16 find the
    maximum number 16

Finally, by the application of hyperrule three and
consistent substitution, the metanotion MAX is synthe-
sised to the value 16. Thus this branch of the tree
terminates and the value of MAX is propagated up the
tree. Branch 1b is the output of the algorithm which is
the maximum number is 16. A rule matching this
hypernotion is not shown, since in practice, this branch
is considered to be the output of the algorithm.

4. CASE STUDY OF TWO-LEVEL
   GRAMMAR PROGRAM DEVELOPMENT

Having shown how TLG programs work, we now show
how to develop a TLG program. The problem we will
consider is sorting a list of numbers using a merge sort.
The TLG program for an algorithm can be considered
as a functional description of that algorithm, with the
metanotions being the arguments or variables in the
functions and the hyperrules defining the functions. In a
TLG program the start rule can be considered as a high-
level specification of the algorithm. The hypernotions on
the right-hand side of the start rule elaborate the
algorithm further and define the output. In this case the
high level specification is to sort a list of numbers, and
the algorithm to be used is the merge sort. From
the specification, we formally define the metanotions (vari-
ables) the TLG program will use as follows:

NUMBER::INTEGER
NUMBER_LIST::(NUMBER)+.
NUMBER_LIST_OPTION::{NUMBER}*.  
EMPTY::.
RESULT::false; true.

The metanotion NUMBER defines an integer,
NUMBER_LIST defines a non-empty list of numbers and
NUMBER_LIST_OPTION defines a (possibly empty) list
of numbers. RESULT is used as a boolean flag.
The start hyperrule, representing the highest level of
abstraction in the functional description of the algorithm,
is:

1. sort list of numbers NUMBER_LIST1:
   merge-sort NUMBER_LIST1 into
   NUMBER_LIST2,
   the sorted list of numbers is
   NUMBER_LIST2.

The metanotions NUMBER_LIST1 and NUMBER_LIST2
represent the same root metanotion – namely
NUMBER_LIST. The distinction is made to avoid the
consistent substitution policy. Thus in the application of
the rules, NUMBER_LIST1 and NUMBER_LIST2 repres-
ent different lists of numbers. The first hypernotion
specifies the algorithm and will synthesise the metanotion
NUMBER_LIST2 as the list of sorted numbers. The second
hypernotion is the output of the program.

The next step is to define another hyperrule with the
left-hand side hypernotion matching the first hypernotion
of hyperrule 1. Since we are developing a functional
program, the first step will be to put in the termination
condition. Thus the second hypernotion we derive is:

2. merge-sort NUMBER_LIST1 into
   NUMBER_LIST2:
   if NUMBER_LIST1 are sorted then
   assign NUMBER_LIST1 to
   NUMBER_LIST2
   otherwise
   divide NUMBER_LIST1 into
   NUMBER_LIST3 and NUMBER_LIST4
   endstmt
   merge-sort NUMBER_LIST3 into
   NUMBER_LIST5 endstmt
   merge-sort NUMBER_LIST4 into
   NUMBER_LIST6 endstmt
   merge NUMBER_LIST5 with
   NUMBER_LIST6 giving
   NUMBER_LIST2
   endif.

As can be seen, this step of the algorithm terminates if
the list of numbers is already sorted; otherwise the
algorithm divides the numbers into two sub-lists and sorts these, and then combines them to form the sorted list. Here we see a recursive application of this rule applied to the two sub-lists created. The ‘then’ part of this rule serves as the termination condition.

The next step is to develop a set of rules for dividing the given list into two sub-lists and a set of rules to merge the two sub-lists. Dividing the given list into two sub-lists involves the special cases of the list containing one number, the list containing two numbers and the general case with the list containing more than two numbers. If the list contains only one number then the second sub-list contains no numbers (i.e. empty list). If the list contains two numbers then the first sub-list gets the first number and the second sub-list the second number. If the list contains more than two numbers then the sub-lists are formed by putting the first two numbers in the first and second sub-lists respectively. The general case is applied recursively to form the sub-lists. The hyperrule for the general case contains metamotions NUMBER_LIST2 and NUMBER_LIST_OPTION which are synthesised by the recursive application of this rule. The divide rules are then:

3. divide NUMBER into NUMBER and EMPTY: EMPTY.
4. divide NUMBER1 NUMBER2 into NUMBER1 and NUMBER2: EMPTY.
5. divide NUMBER1 NUMBER2 NUMBER_LIST1 into NUMBER1 NUMBER_LIST2 and NUMBER2 NUMBER_LIST_OPTION: divide NUMBER_LIST1 into NUMBER_LIST2 and NUMBER_LIST_OPTION.

Having developed rules to divide the lists, we now develop rules for merging two lists. Two special cases are that one or the other of the lists is empty, and the general case is that neither list is empty. The rules for merging are given below:

6. merge EMPTY with NUMBER giving NUMBER: EMPTY.
7. merge NUMBER with EMPTY giving NUMBER: EMPTY.
8. merge NUMBER1 NUMBER_LIST_OPTION1 with NUMBER2 NUMBER_LIST_OPTION2 giving NUMBER3 NUMBER_LIST2: if NUMBER1 greater than NUMBER2 then assign NUMBER2 to NUMBER3 endstmt merge NUMBER1 NUMBER_LIST_OPTION1 with NUMBER_LIST_OPTION2 giving NUMBER_LIST2 otherwise assign NUMBER1 to NUMBER3 endstmt merge NUMBER_LIST_OPTION1 with NUMBER2 NUMBER_LIST_OPTION2 giving NUMBER_LIST2 endif.

The last rule for the general case splits the two lists being merged into the first number and the remaining numbers. Then we have an ‘if-then-else’ type of compound statement, where the first number of the first list is compared to the first number of the second list, and the greater number is assigned to NUMBER3. The metanotion NUMBER_LIST2 is synthesised by the recursive application of the general case of the divide rule. The assignment rule (actually an identity function) is quite straightforward and will not be shown; the assumption being that the control flow statements and assignments can be considered as primitives and are the framework of the language. In this example, we are more interested in the higher level functions of the algorithm.

Although we will not consider the rules for the conditional statement itself, we will show the rules to ‘evaluate’ the condition associated with the conditional statement; that is, the rules for checking whether the list is sorted.

9. evaluate condition NUMBER are sorted giving true: EMPTY.
10. evaluate condition NUMBER1 NUMBER2 are sorted giving RESULT: evaluate condition NUMBER1 less than NUMBER2 giving RESULT.
11. evaluate condition NUMBER1 NUMBER2 NUMBER_LIST are sorted giving RESULT: if NUMBER1 greater than NUMBER2 then assign false to RESULT otherwise evaluate condition NUMBER2 NUMBER_LIST are sorted giving RESULT endif.

It should be noted that the assignment rule used in this example serves to return the result of a function call, and does not assign a value as a side effect in the sense of an imperative programming language. We also mention that metamotions could be considered as variables; however the variables are actually function arguments and not variables as used in imperative languages. A program is said to be referentially transparent if it prohibits assigning different values to the same named variable during the same run. Referential transparency is a requirement of functional programs. Therefore, two-level grammar can be used to develop ‘pure’ functional programs. It is possible to define structured statements such as ‘while’ and ‘case’ statements using recursive functions (see Edupuganty and Bryant for some of these definitions). The power of the two-level grammar is its ability to define high-level functions in a clear and concise way.

5. COMPARISON OF TWO-LEVEL GRAMMAR PROGRAMS WITH LISP AND PROLOG PROGRAMS

In this section we compare the TLG programs of the previous examples with equivalent ‘pure’ Lisp and Prolog programs. The comparison is made with respect to (1) readability, (2) correspondence to the function definition, (3) data structure (lists) manipulation, and (4) the capability to define the domain of the function arguments.

The function definition and the TLG program for the palindrome problem are given in Fig. 1. The corresponding Lisp and Prolog programs are given in Fig. 3.
Lisp program

(defun palindrome(string)
  (cond ((null string) t)
        ((= (length string) 1) t)
        ((equal (car string) (car (last string)))
         (palindrome (all but last (cdr (string)))))
        (t (all but last (str)
           (reverse (cdr (reverse str))))))

Prolog program

palindrome([]).
palindrome([X|Tails]):= palindrome(Tails), palindrome([X|Y]), palindrome(Y).

Figure 3.

From the programs given in Figs 1 and 3, it is apparent that the TLG program most closely corresponds to the functional definition of the problem. We also claim that the TLG program is more concise and readable, since additional selector functions are not required due to the pattern matching capability. Although Lisp is a functional programming language, the code in Fig. 3 does not correspond to the function definition. This is primarily due to its syntax, but the important observation is that additional selector functions (list operators) are required to manipulate the data structure. The same is true of the Prolog program. The fact that both these languages allow operations only on the `head' of a list necessitates additional selector functions. In both the Lisp and Prolog programs the variables used in the programs are untyped, whereas the TLG program, using the metarules, has the capability of specifying the domain of the variables. The capability of TLG programs to formally define function domains in a straightforward manner is a significant advantage over other functional programming languages.

As an additional example, consider the linear search problem of Section 3.4 and the three programs for solving that problem. The function definition and the TLG program are given in Fig. 2 and the Lisp and Prolog programs are given in Fig. 4. It is our opinion that the TLG program is the most readable of the three. All the programs compare favourably with respect to correspondence to the function definition. The discussion on the domain specifications is general to TLG programs and applies in this case too.

6. SUMMARY AND CONCLUSIONS

We have shown how two-level grammar can be used as a functional programming language and as a formalism for writing recursive function definitions. The programs are easy to read due to the natural language vocabulary which may be used freely. The elegance of the TLG programs can be attributed to their close correspondence to the functional definition of the problem. The TLG programs are referentially transparent and side effect free. The main result is that the TLG formalism can be used as a functional programming language. In this paper, two-level grammars have been given for
(1) the palindrome problem,
(2) finding the maximum of a list of numbers, and
(3) sorting a list of numbers using the merge sort.

Examples have been used to illustrate the differences between a programming language based on TLG and equivalent Lisp and Prolog code. We have also shown how TLG programs can be developed using top-down step-wise refinement. We have described a method of interpreting two-level grammars to make all these algorithms implementable.

The advantages of TLG programs are summarised below:

1. By using a natural language vocabulary to express recursive function definitions, TLG programs are virtually self documenting. Additional readability is achieved from the direct relationship of TLG programs to the recursive function definitions.

2. Referential transparency, a characteristic of functional programming languages, is achieved since there are no assignment statements. The identifiers (program variables) used in the TLG programs only serve as arguments to functions.

3. The metarules can be used to define specific or polymorphic data types and data structures can be defined to suit the application. Hyperrules can be used to define constructor and selector functions to access the defined data structures (as in our examples). The abstraction made possible by the metarules also allows formal specifications of the domains of the function arguments, which is a significant advantage over other functional programming languages that do not have features for data abstraction. Static checking of data types is also made possible (e.g. to catch mis-spelled or undefined domains).

4. The programs are non-procedural and it is easy to develop TLG programs in a top-down manner from the problem specification. Using a top down design the problem can be decomposed into sub-problems or sub-
functions. Rules for the smaller sub-problems can then be developed in a bottom-up manner. Grouping the rules would, we claim, aid the development process, and the groups of rules can be considered as modules. Thus 'good' software design techniques can be applied in developing TLG programs. Also, because of the flexibility of the two-level grammar notation, algorithm developers may create their own control structure definitions best suited to the application using their own syntax.

Our extended research goals are: (1) the incorporation of higher order functions into two-level grammar functional programs, (2) the implementation of languages such as Lisp and Prolog within the TLG framework, (3) a syntax-directed editor to facilitate writing two-level grammars, and (4) to investigate the possibilities for varying the application order to include a call-by-name order.

Acknowledgement

The authors are grateful to Dr Der-Yun Yeh of Arizona State University for his helpful comments.

REFERENCES