TRANSFORMATICNAL GRAMMARS: THEIR APPLICATIONS AND IMPLEMENTAIION

By

ALAN LYNN ROBERTSON

Bachelor of Science in Electrical Engineering

Oklahcma State University

Stillwater, Oklahoma

1976

Submitted tc the Faculty cf the
Graduate College of the
Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
May, 1978

TRANSFORMATIONAL GRAMMARS: THEIR APPLICATIONS AND IMPLEMENTATION

Thesis Approved:

J. E. Hedrick

Thesis Adviser

J P Chandler

James R VanDorn

Norman N Durham

Dean of the Graduate College

PREFACE

This thesis is a description of an implementation of an experimental transformational grammar system developed for use in a translator writing system. Transformational grammars are useful in compiler-writing because they are a powerful mechanism for the manipulation of (syntax) trees. Thanks are due to Dr. J. P. Chandler, Dr. James R. Van Doren, and especially to Dr. G. E. Hedrick for their suggested improvements of this thesis. The author would also like to acknowledge the support of the National Science Foundation for sponsoring this research under grant NSF-MCS576-06090. I would also like to thank my wife, Laura, for her understanding and encouragement throughout the entire project.

TABLE OF CONTENTS

TABLE

LIST OF FIGURES

CHAPTER I

AN INTRODUCTION TO TRANSFORMATIONAL GRAMMARS

Introduction

The major goal of this thesis is to describe an experimental implementation of a transformational grammar system developed by the author. This transformational grammar system is designed to be used as a basis for a translator writing system. Since transformational grammars have not yet been applied to computer science to any great extent, a secondary goal of this thesis is to illustrate how the formalism of transformational grammars can be applied to some practical compiler construction problems.

Transformational grammars are not generative string grammars like the more familiar regular, context-free, context-sensitive and unrestricted grammars of the Chomsky hiearchy. That is, the rewriting rules (productions) of transformational grammars do not specify the generation of sets of strings. Instead, transformational grammars specify a set of transformations or structural changes to be performed on trees in some tree domain. Moreover, the rules of conventional string grammars are generally constrained to operate only upon strings which can be produced by other

rules in the grammar. When using transformational grammars, the rules are often applied both to (sub) trees produced by other rules in the grammar, and to trees produced by some external tree generation mechanism. One such "external" tree generator is an LR(k) parser which produces a parse tree to be used as input tc the transformaticnal grammar [6]. Under these circumstances, the output of the transformational grammar is often referred to as an "abstract" syntax tree. This terminology is used to distinguish it from the derivatien tree resulting from the concrete syntax of the language defined by a context-free grammar. An abstract syntax tree is a tree whose contents and structure are based upon a strict derivatien tree. It differs from the derivatien tree in that the information in a derivation tree is crganized according to syntactic necessity, not semantic content. Typically, nodes for single productions may be removed, much syntactic puncuation (";", "(", ")", ",", etc.) without any semantic connotations may be remcved, and the relationships between subtrees may be modified to suit semantic rather than syntactic necessity. At one extreme, the transformaticnal grammar could translate an input derivation tree into a form very nearly suitable for direct interpretation by a computer (machine-code) or a ccmputer program (an interpretive code). A somewhat more modest application can be found in the creation of an abstract syntax tree in some form convenient for use by the code-generatien routines of a compiler.

Some Applications of Transformational Grammars

Transformational grammars (T-grammars) are a very powerful system with a broad range of potential applicability. Several areas where they could prove useful are listed below.

1) Much semantic analysis within compilers is essentially syntactic in nature (i.e.: involves the manipulation of text), but is beyond the capacity of context-free grammars alone [5]. Transformational grammars are a useful formalism for expressing this syntactic type of work in a way that conventional context-free grammars cannot. For optimizing compilers, abstract syntax trees are the preferred internal representation of the source program.
2) In some applications with simple compile-time semantics, such as FORMAT-denotations in ALGOL 68 or FORMAT statements in FORTRAN, it can be attractive to interpret the abstract syntax tree produced by a T-grammar directly. This would allow the entire compilation process for these sublanguages to be performed by a set of well-understood formal techniques. This is of special interest for those cases where an interpreter is used as a means of formal semantic definition [13].
3) Many complex forms of algebraic expression

simplifications can be expressed in terms of a T-grammar. The work described in this thesis originally stemmed from the author's implementation of such a T-grammar based simplification technique.

4) Many useful translations which cannot be expressed using other widely-known techniques, such as syntax-directed translation, are easily performed using simple T-grammars.

Macro Languages, T-Grammars, and General Replacement Systems

Transformational grammars are quite similar to string macro languages in many respects. In the usual case, neither a macro language nor a transformational grammar is used as a generator of strings. In both cases, the most important function that the systems serve is the transformation of the input to some other form, not the determination that the given input is a member of some set of acceptable inputs (as is the case with the more conventional grammars). With both macros and T-grammars, it is very difficult to speak meaningfully of the transformation (substitution) rules independently of the set of inputs which they are designed to transform. As a result of these and other similarities, transformational grammars could be regarded rather loosely as a type of macro language for trees.

Both string macros and T-grammars belong to a broad

class of mathematical systems known as general replacement systems. A general replacement system is a set of objects and a set of rules for replacing objects in the set by other objects in the set. These replacements are performed upon an object until no more replacements can be made. The resulting object (if any), is often then described as being in "normal" form. All the systems in this general class, despite their diversity, share many common characteristics. As a result, various features of T-grammar systems will be presented by analogy to their corresponding features in the more familiar string macro replacement systems. For example, the simplest form of macro specifies that all substrings of a given form are to be unconditionally replaced by some other string. An example of such a macro definition or rule is: abcd => cde. The symbol "=>" is used to denote replacement, and is read as "goes to" or "is replaced by". An application of this rule would cause an occurance of "abcd" in some string to be replaced by the string "cde". This transformation process takes place in two steps. First, a string pattern match takes place. This match determines the substring in the given string to which the transformation or substitution is to be applied. Given the string "abcdefg", the pattern match proceeds by determining that the substring "abcd" matches the left hand side (LHS) of the given rule. For this pattern matching, the LHS is said to imply a set of strings which are said to "match" the given LHS. For this case, the set of strings

which is implied is simply {"abcd"}. The second step is the substitution or transfcrmation step. In this step, the substring which was found to match the LHS of the rule is replaced by (transfcrmed into) the string implied by the right hand side (RHS) of the rule. This would ccnsist cf substituting "cde" for "abcd". The net effect of this transformation would be tc yield the string "cdeefg". A very similar process takes place when applying a rule of a T-grammar to a given input tree. A sample rewriting rule from a T-grammar is given in Figure 1. This rule specifies that a subtree of the form of the LHS is to be replaced by a subtree of the form of the RHS.

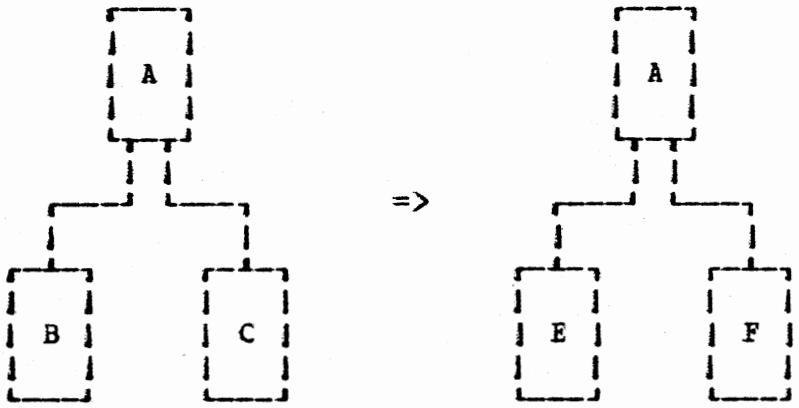

Figure 1. A Simple Transformation Rule

As in the case of macros, this transformation takes place in two steps. The steps take place in the same manner as before, except that now the pattern matching algorithm tries to match a tree to a subtree instead of trying to match a string to a substring. During the transformation step, instead of replacing the original substring by the transformed string, a T-grammar would replace the original subtree by a forest of subtrees (possibly containing only one subtree).

One can also define "parameterized" general replacement systems. An example of a parameterized macro rule is ADD(s1,s2) => s1 + s2. In this rule, the symbols s1 and s2 are parameters to the rule. As such, they may represent any string. When a reference to this macro is detected, the arguments supplied are "evaluated" and bound to the parameter names. The evaluation of the parameters consists of the evaluation of any macro invocations contained in them. When a macro is evaluated the values of the strings which were supplied as arguments are available for use in the RHS of the rule. An application of this rule would transform the string "ADD(X,Y)" into the string "X+Y" and two applications would transform the string "ADD(A,ADD(B,C))" into the string "A+B+C".

Correspondingly, one can also consider parameterized T-grammar transformation rules in which the parameters are subtrees (or forests of subtrees). It is this notion of parameters which gives T-grammars much of their flexibility

and power. The introduction of parameters into the rules affects the cardinality of the set of subtrees which match the LHS of a rule. A countably infinite set of subtrees will match any T-grammar rule with at least one subtree parameter. An example of a T-grammar rule with parameters is given in Figure 2. In this rule, the parameters are the (unenclosed) symbols t1 and t2. If this rule is applied twice to tree A in Figure 3, tree B from Figure 3 is the result. The first application of the rule results in a tree which matches the LHS of the rule. This intermediate tree is then transformed according to the RHS of the rule, yielding tree B in Figure 3. In general, this type of re-application process can be quite useful. In part, it is this ability to transform a subtree more than once which allows T-grammar systems to perform more sophisticated types of tree manipulations than are possible with techniques such as syntax-directed translations.

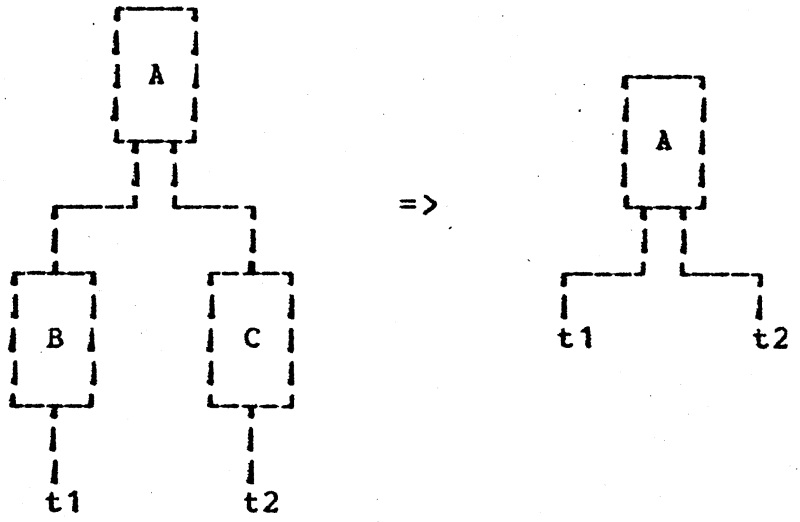

Figure 2. A Transformation Rule with Parameters

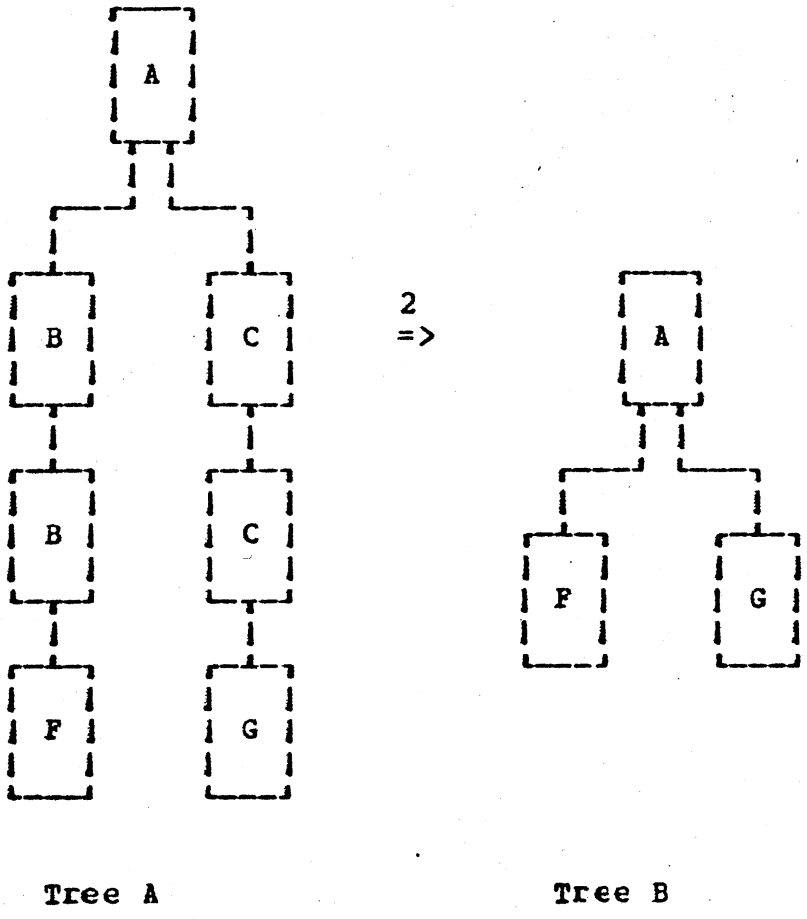

Figure 3. An Example Tree Transformation

A Review of the Literature

The notion of transformational grammars has been associated historically with linguistics and the modeling of natural (human) language. The notion of transformation rules is attributed to Chomsky [4]. Much of the computer science literature in the area of transformational grammars is theoretical in nature and is often more directly associated with the concept of tree automata than is the author's work. In order to preserve mathematical simplicity, some studies deal with grammars whose rules are of a restricted form. These grammars are capable of performing many of the same transformations as more complex forms, but they require more effort to use. Since the major concern here is with implementation considerations, the author's emphasis differs greatly from that of the majority of the literature. Only relatively recently has the usefulness of T-grammars in computer science been studied to any degree. The primary reference for practical applications of T-grammars to computing, particularly compiler construction, is De Remer [6]. The work described here parallels that of De Remer in many ways. Unless otherwise stated, the terminology used here is the same (or similar) to De Remer's. De Remer gives an excellent introduction to T-grammars for compiler-writers and contains several detailed examples of their applications to various compiler-writing problems. Aho and Ullmann [1,2], discuss a

simple form of tree manipulation, namely syntax-directed translations. Syntax-directed translation systems can be used to perform some of the more basic operations that T-grammars perform, but are by no means as powerful a tool as T-grammars. Rosen [15] proves certain theorems concerning a general class of tree manipulations known as subtree replacement systems. This paper is a good introduction to some of the more rigorous definitions and properties of general tree-manipulation systems (including T-grammars). Rounds [16] considers a form of T-grammar somewhat closer to the type used here, except that he considers only top-down transformations. His definitions are precise and several examples are given and explained in detail. Even though he treats only top-down tree automata, this work provides a solid theoretical base from which T-grammars may be considered. Ginsburg and Partee [7] provide a set of definitions and examples of T-grammars as applied to the study of natural languages. An effort was made to make the linguistic concepts clear and accessible to the mathematician and/or computer scientist. The transformations considered in this paper are essentially top-down transformations as opposed to the bottom-up approach adopted in the author's research. The form of "structural match" and "structural change" statements used by Ginsburg and Partee, and in most linguistic literature, is only remotely related to the forms used by De Remer, Rosen, Rounds, and this author. In linguistics, T-grammars

are usually used to indicate the semantic equivalance of two or more forms of a sentence. T-grammars are used for this purpose in artificial language translation, but such semantic redundancy is relatively rare in artificial languages.

CHAPTER II

SOME EXAMPLE T-GRAMMARS AND TRANSFORMATIONS USING THEM

Notational Conventions

Before considering a series of T-grammar systems it is desirable to establish some notational conventions with regard to a linearized representation for trees and T-grammar rules. The representation chosen here is identical to a linearized prefix form described in [6]. The representation is derived from the standard prefix Polish notation for arithmetic expressions. Since this prefix notation is to be used to express general n-ary trees, it is necessary to bracket the subtrees in some fashion to indicate which subtrees are descendants of which other subtrees. Angle brackets ("<" and ">") are used for that purpose. For example, the linearized representation of an (abstract) syntax tree for the expression A+B*(C+D) would be:

+ A < * B <+ C D >>.

In this tree, the root node is '+' and its two subtrees are 'A' and < * B <+ C D>>. The node '*' in turn has two descendants, 'B' and <+ C D>. The node '+' has two subtrees also. They are simply 'C' and 'D'. The corresponding

standard prefix Polish representation for the same expression would simply be: + A * B + C D. This linearized representation can be defined by an algorithm for printing the tree. This printing is done during a standard pre-order traversal of the tree. In order to indicate unambiguously how many subtrees any given node has, the printing of a subtree and its offspring are "bracketed" according to the following rule: IF a subtree is not the entire tree or a leaf THEN print a "<" , the subtree (in preorder), and a ">" ELSE print only the subtree (in preorder). In terms of the usual binary expression tree, the brackets surround an operator and all of its operands (unless the operator is at the root of the tree). Using these rules, the trees of Figure 3 of the first chapter would be:

Tree A: 'A' < 'B' < 'B' 'F'> > <'C' < 'C' 'G' > >.

Tree B: 'A' 'F' 'G'

Since a T-grammar's transformation rules are essentially tree-structured, similar conventions have been adopted here for representing them. In this case though, there are also forests of subtrees to consider (such as may occur in RHS of a rule). To allow these forests to be represented, the following convention will be followed: A RHS of a rule which is not enclosed by brackets shall be considered a forest of subtrees rather than a single subtree. For further clarification, refer to the CF grammar in Appendix A.

Classes of T-grammar Systems

With these conventions established, several classes of T-grammars may be defined. The first system which will be considered is a non-parameterized, simple substitution T-grammar system. The rule shown diagrammatically in Figure 1 of the first chapter is a rule from such a system. It could be represented linearly by the string:

'A' 'B' 'C' => <'A' 'E' 'F'>.

Note the the RHS must be explicitly enclosed in angle brackets to indicate that the nodes 'E' and 'F' are descendants of 'A' rather than its siblings, as would be the intent if the brackets were omitted.

The next class of systems to be presented is the class of simple parameterized T-grammar systems. The rule in Figure 2 of the first chapter is an example of such a rule. It would be represented linearly as:

'A' <'B' t1> <'C' t2> => <'A' t1 t2>.

The node names ('A' and 'B') are surrounded by apostrophes and the parameters (t1 and t2) are not enclosed by apostrophes. This is to distinguish a node name of 't1' from the rule parameter t1. Using only T-grammar rules of this form, many useful functions may be performed. Several practical examples of this type may be found in [6]. One such system from [6] is a single rule system. This rule is:

'+' t1 <'+' t2 t3 > => <'+' <'+' t1 t2> t3>.

This rule, which has three parameters, performs a well-known

optimizing transformation which minimizes the number of registers required to compute nested sums in algebraic expressions. It would transform the abstract syntax tree for an expression like A+(B+(C+D)) into the syntax tree for the algebraically equivalent expression ((A+B)+C)+D. The latter form uses only one register (temporary) for its computation whereas the original form uses three registers.

A simple extension to the notation described thus far allows one to "factor" common subtrees out of similar patterns. This is useful since it allows the rules to be written more concisely and can speed up the subtree pattern matching process considerably. The notation for this device is the same as used in [6]. This mechanism is specified by the use of an "alternative" (or) operator. This operator specifies that a subtree will match a given sub-pattern if the subtree matches any of the alternatives which are separated by "|"'s. All subtrees which belong to the same alternative list are grouped with parentheses. This is similar in syntax to a convention used to represent "factored alternatives" in context-free grammars. An example of such a factored T-grammar rule would be:

'A' ('B' | <'B' t1>) 'D' => <'A' ('C' | <'C' t1>) 'E'>.

This rule could be used to replace the following two rules:

'A' 'B' 'D' => <'A' 'C' 'E'>,

and 'A' <'B' t1> 'E' => <'A' <'C' t1> 'E'>.

When several factored subtrees are present in a rule, their meaning may become ambiguous. For example, given the rule:

```
'A' ( <'B' t1> |<'C' t2> ) ('D'|'E') =>
        <'B'('F'|'G') (t1|t2) >,
```

it is difficult to establish mechanically that the order of each of the alternative-lists in the LHS and RHS are reversed with respect to one another. That is, the first alternative-list in the LHS of the rule corresponds to the second alternative-list in the RHS. More complex situations also arise. For example, if the rule

```
'A'('B'|'C')('D'|'E') => <'A'('I'|'M')('L'|'M')('N'|'O')>
```

is to be meaningful, the correspondences between the LHS and RHS alternative-lists must be defined. The approach taken here is simply to number the alternative-lists in the LHS and RHS. Those alternative-lists in the RHS which have the same number as some alternative-list in the LHS are said to "correspond" to that alternative-list. Given such a numbering convention, one interpretation of the previous rule would be:

```
'A'(1 'B'|'C') (2 'D'|'E') =>
    <'A' (1 'L'|'M') (1 'L'|'M') (2 'N'|'O')>.
```

Each alternative-list in the RHS of a rule must correspond to exactly one alternative-list in the LHS, but an alternative-list in the IHS need not have any alternative-list to which it corresponds, or may have many lists in the RHS to which it corresponds. Since no two alternative-lists in the LHS of a rule may have the same number, it is convenient to omit their explicit numbers and assume that they are implicitly numbered starting from one in a left-to-

right manner. Using such an implicit numbering scheme, the rule above could be re-written as:

'A' ('B'|'C') ('D'|'E') =>

<(1 'L'|'M') (1 'I'|'M') (2 'N'|'O')>.

Such an implicit numbering scheme prevents any two alternative-lists in the LHS of a rule from accidentally being assigned the same number.

The next increase in power (and complexity) of this method involves the use of a notation which will allow a regular set of trees to be described. This notation uses the asterisk to denote zero or more subtrees of a given form. This asterisk operator is sometimes referred to as the Kleene closure operator [12]. It is a suffix operator, and as such, it follows the subtree to which it applies. For example, <'A' 'B'*> denotes the infinite set of trees

{<'A'>, <'A' 'B'>, <'A' 'B' 'B'>, ... }.

By the same conventions, <'A' <'B' 'C'>* > denotes the set of trees

{<'A'>, <'A'<'B' 'C'>>, <'A'<'B' 'C'><'B' 'C'>>, ... }.

This mechanism can extend the class of transformations which can be described by a T-grammar in much the same manner as did parameters in the rules. Unfortunately, this powerful technique presents some difficult implementation problems. If this technique is implemented in its fullest generality, some rules can be written which are difficult to interpret, and some may be ambiguous. One such difficult rule is:

'A' <'C' t1>* <'C' t2> <'C' t3> => <'A' <'C' t2>>.

The effect of this rule is to replace a forest of at least two subtrees by the next-to-the-last subtree in the forest. Here, when the * operator is used, the problem is one of computational complexity. It can be quite time consuming to determine that the LHS of a particular rule does not match a given subtree. A simple-minded pattern matching facility might try various replication values for the *-repeated subtree until one causes a match for the entire pattern, or fails to match the given tree. In other words, it would allow the subtree replicated by the * (<'C' t2>) to be a forest of zero, one, two, and more trees successively, until it exceeds the number of such subtrees actually present in the given subtree. This repetitive, trial-and-error aspect of the pattern matching can be likened to automatic top-down syntax analysis with backup. In other words, like most other combinatorial problems, it can be extremely slow, especially if more than one * is present in a pattern at the same level, or in a nested usage such as 'A'<'B' 'C'*>*. An approach analagous to finite state automatons can be used to recognize that a given subtree matches a given pattern, but many semantic connotations are lost by such an approach. There are other implementation problems associated with the closure operator, but they will not be considered further here.

Since the closure operator seems to provide a desirable facility for dealing with "bushes" (trees with arbitrarily many subtrees), but also provides the implementor with many

more headaches, it is reasonable to ask if there is another mechanism which will perform many of the same functions at a minimal implementation cost. One such mechanism may be found in a T-grammar rule parameter which allows subtree parameters in T-grammar rules to represent non-empty forests of subtrees. This type of parameter is an extension over the single-subtree parameters used in [6] and elsewhere. The convention used here to represent single-subtree parameters is t"integer". Some examples of single subtree parameters would then be t1, t2, and t99. For forest-valued parameters, the letter "s" is used instead of "t". This convention yields names like s1, s2, and s301. The ability to replace the * by s-type parameters efficiently is heavily dependent upon knowledge about the domain of trees input to the rules. For example, 'A' 'B' 'B'* => <'A' 'C' 'C'*> may be replaced by the following two rules:

'A' ('B' | 'B' s1) => <'A' ('C' | 'X' s1) >,

and 'X' ('B' | 'B' s1) => ('C' | 'C' <'X' s1>).

In these rules, 'X' is some symbol distinct from all other symbols in the tree. This replacement is valid only if all subtrees of the form <'A' 'B' s1> in the domain are also of the form <'A' 'B'*>. The single substitution which is represented by the rule with a * is effectively replaced by a recursive re-application of a set of rules. The T-grammar rules given above function by splitting off the 'B' nodes one at a time and processing each one without considering the nature of the entire subtree at once. This permits much

greater implementation simplicity, but for those rare cases when lookahead is necessary, it may require a complex set of rules to simulate it. This enables the more common rules to be implemented without the overhead of the lookahead mechanism implied by the * operator. It should be noted that this form of T-grammar is capable of generating any recursively enumerable set. For an informal proof of this fact, see Appendix C. This implies that any problem which can be solved by any other computing mechanism can also be solved by this T-grammar formalism.

A Practical Example

An example of the practical usefulness of T-grammars can be found in the compilation of PLI structure and array references. According to the PL/I Language Reference Manual for the IBM Checkout and Optimizing Compilers [11], the subscripts of an array of structures containing arrays may be placed either together or apart. This ability to specify the subscripts between any pair of qualifiers desired is a difficult feature for a compiler to implement correctly. A sample PL/I program which uses these subscript forms is shown below.

```
DECLARE
  1 A(3),
    2 B,
      3 C(4) FIXED BINARY(31,0);

A(1).B.C(2)=0;
A.B(1).C(2)=0;
A(1).B(2).C=0;
A(1,2).B.C=0;
A.B(1,2).C=0;
A.B.C(1,2)=0;
```

To determine which data item is being referred to by a given subscripted/qualified reference, it is necessary for the compiler to gather all of the qualifiers together to form the qualified name. In the example above, this "qualified name" would be "A.B.C". The compiler then looks up the reference in its symbol table and determines the number of (inherited) dimensions (two in the example) that the data item actually has. This is compared to the total number of subscripts supplied in the reference. If the number supplied does not match the number declared, an appropriate error message is issued. This type of separation of subscripts and qualifiers cannot be performed by most other syntax tree manipulation systems (such as syntax-directed translations). Conventional PL/I compilers may implement special symbol table mechanisms solely to handle this type construction. However, a five-rule T-grammar can achieve the same effect without any special case provisions. A context-free grammar which generates all syntactically legal subscripted and/or qualified references is presented below. A T-grammar which separates the qualifiers and the subscripts conveniently is also

presented.

Context-Free Grammar for PL/I Variable References

```
REFERENCE: IDEN;
           REFERENCE, DOT, IDEN;
           SUBPARM-REF, DOT, IDEN;
SUBPARM-REF: REFERENCE, '(', EXPR-LIST, ')'.
EXPR-LIST: EXPRESSION;
           EXPR-LIST, COMMA, EXPRESSION.
```

Corresponding T-grammar for PL/I Variable References

```
'EXPR-LIST' < 'EXPR-LIST' s1> 'COMMA' t2 =>
    <'EXPR-LIST' s1 t2>.
'REFERENCE' t1 => <'REF' <'QUALS' t1> 'SUBPARMS'>.
'REFERENCE' <'REF<'QUALS' s1> s2> 'DOT' t3 =>
    <'REF' < 'QUALS' s1 t3> s2>.
'SUBPARM-REF' <'REF' t1 t2> '(' t3 ')' =>
    <'REF' t1 <'ADD-SUBPARM' t2 t3>>.
'ADD-SUBPARM' ('SUBPARMS' | 'SUBPARMS' s1) t2 =>
    <'SUBPARMS' (t2 | s1 t2) >.
```

It should be emphasized that this T-grammar is designed to transform the derivation trees produced by the context-free grammar above. After being transformed by the T-grammar, an abstract syntax tree for a variable reference has two subtrees, one (named 'QUALS') with all the qualifiers specified as its descendants, and one (named 'SUBPARMS') with all subscripts and parameters specified as its descendants. Note that 'REF', 'QUALS', 'SUBPARMS', and

'ADD-SUBPARM' are new node names introduced by the T-grammar for its internal use. The first rule in the T-grammar effectively transforms the (left) recursion inherent in the non-terminal "EXPR-LIST" into a linear (iterative) form. The second rule transforms the initial identifier in a reference into a subtree with a left descendant containing the identifier as the only subtree of a 'QUALS' (qualifiers) node, and a right descendant containing the list of subscripts for this reference (initially empty). The third rule is applied when a reference is qualified by a subsequent qualifier. This new qualifier is attached as the last subtree of the 'QUALS' node for the reference. The fourth rule is applied whenever a reference is subscripted (or supplied with parameters). This rule causes the subscripts to be attached as the last subtree of the 'SUBPARMS' node for the reference by forcing application of the last rule. Introduction of the node named 'ADD-SUBPARM' into the tree forces actions in a manner analogous to those which are forced by entering a new state in a tree automaton. When "in" this "state", the T-grammar is caused to examine a set of transformations which are appropriate under these circumstances. The introduction of this new node name into the tree forces the basically bottom-up transformer to perform transformations in a top-down fashion. The fifth (and last) rule in the T-grammar has two subcases, one subcase for the first subscript group in a reference, and one subcase for subsequent subscript groups.

If "i" and "j" represent identifiers, and "e" and "f" represent expressions, then the string "i(e).j(f)" would be a legal reference form. The abstract syntax tree for this reference would be:

<'REF' <'QUALS' <'ID' 'i'><'ID' 'j'>>
<'EXPR-LIST' <'EXPRESSION' 'e'>>
<'EXPR-LIST' <'EXPRESSION' 'f'>> > >.

The abstract syntax tree would be nearly identical for all semantically (logically) equivalent reference forms that can be specified, regardless of the positioning of the subscripts. This additional uniformity also serves to ease the compiler-writer's task by reducing the total number of cases which must be considered. This T-grammar effectively groups the qualifiers together ("i" and "j") and separates out the subscripts ("e" and "f") for later analysis. This frees the compiler writer from having to design special-case code to deal with the matter. This particular example was chosen since it it a rather practical example mentioned specifically in [2] as being beyond the power of syntax-directed translations alone. For further practical examples from PL/I and other languages, the reader is referred to [6], and to the example in Appendix B.

CHAPTER III

IMPLEMENTATION OF A T-GRAMMAR TREE TRANSFORMER

Overall Description

The tree transformer implementation described here is a bottom up transformer. That is, it transforms the tree beginning at the leaves and working its way up to the root of the tree. The most convenient approach to use in this case involves transforming the tree as it is being built. This does not imply that a top-down parsing technique may not be used to generate the tree. It is only necessary that the (syntax) tree produced by the external tree generator be constructed in a bottom-up fashion. This may be easily done in conjunction with any parsing technique [6,2]. Due to the specialized nature of the transformations, they are accomplished by an interpreter rather than by specially generated code which could be compiled or linked into a working transformer. An interpretive approach generally implies a large decrease in memory requirements and a relatively small time overhead when compared to generated machine or high-level language code. The time overhead is small because the primitive operations being performed are

relatively complex compared to the overhead of the interpretation process.

Basic Logical Structure

The tree transformer consists of three major modules: a control module, a tree pattern matcher, and a tree generator (the actual transformer). The controlling routine scans the LHS's of the rules and invokes the pattern matching module to determine whether the current subtree matches the given pattern. If a match occurs, the control module invokes the tree constructor module to construct a new subtree according to the RHS of the given rule. After creating the new tree, the tree constructor invokes the control module to determine whether the constructed tree should be further transformed according to some rule in the T-grammar. In the actual implementation, the control module is named "translate", the tree constructor is named "transform tree", and the pattern matching module is named "pat matches tree". Their direct calling relationships are shown below in Table I.

TABLE I

DIRECT CALLING RELATIONSHIPS OF THREE KEY MODULES

Routine name	Procedures called	Called by
translate	transform tree pat matches tree	transform tree
pat matches tree	pat matches tree	pat matches tree translate
transform tree	transform tree translate	transform tree translate

Note that the routines call upon each other in a highly recursive fashion. The routines "translate" and "transform tree" call upon each other in a mutually recursive fashion, in addition to the simple recursion used in "transform tree" to traverse the output pattern (RHS) of the rule. The use of recursion in these procedures is conceptually simple and straightforward, but when applied to even relatively simple examples, it can prove to be quite difficult to follow in all of its details.

In the process of attempting to match a given subtree to a given LHS (pattern) of a rule, the values of all the subtree parameters must be determined and returned to

control module (translate). Since each rule has a fixed number of parameters associated with it, the parameters are represented internally as a single-dimensional array of pointers to the subtrees that are the actual values of the parameters to the rule. The control module then passes this vector of parameters along to the tree constructor module (transform tree) where they are then available for use in the transformed tree.

The actual algorithm used for matching trees to patterns is not too difficult to derive since it is a straightforward implementation of the following six rules.

1) An empty (sub) pattern matches only an empty (sub) tree.
2) An empty (sub) tree matches only an empty (sub) pattern.
3) If the current (sub) pattern is an "s" parameter node, then the current (sub) pattern matches the given (non-empty) forest of (sub) trees.
4) If the current (sub) pattern is a "t" parameter node, then it matches the current (sub) tree if the patterns right sibling matches the current (sub) tree's right sibling.
5) If the current (sub) pattern node is a named internal node, it matches the current tree node if its name matches the name of the tree node and the siblings and descendants of the current pattern node match the corresponding siblings and descendants of

the current tree node.

6) A choice subpattern matches its corresponding subtree if any of its constituient subpatterns match the given subtree.

Internally, all the node names are assigned consecutive integers for identification. This enables "translate" to easily classify each rule according to the name of the root node of the LHS of the rule. Hence, it need search only a fraction of the rules in order to determine whether any transformation rules apply to a given subtree. This generally produces a noticeable improvement in speed over a single list of rules.

The tree constructor procedure (transform tree) traverses RHS of the rule and constructs a new tree according to the pattern, making copies of the parameter sub-trees when appropriate. After the result tree has been constructed, it calls "translate" (recursively) to further transform the result tree, if necessary.

Subtree Copying Considerations

This entire process is moderately simple until one considers the number of times that a subtree may be copied and recopied during the building of a tree. With this in mind, it seems desirable to minimize the number of unnecessary copying operations that may take place. Copying of subtrees cannot be eliminated altogether because the right sibling (link) of a subtree may be modified by the

tree constructor. An obvious example of the necessity of copying can be found in the following T-grammar rule:

'A' t1 => <'A' t1 t1>.

It seems that whatever subtree that t1 might represent, any attempt to make it into its own sibling would prove disastrous. Given only examples of this kind, it might seem sufficient to copy a subtree if its right sibling (link) is to be modified, and not copy it otherwise. However, since transformations may interact with each other in complex ways, such a strategy will prove to be insufficient. An example of such an inadequacy can be found in the following T-grammar rule:

'X' ('A' s1 | s2) => (<'X' s1 > 'A' s1 s1 | s2).

In this rule, it may seem sufficient to copy the parameter "s1" only for the second instance of s1 in the RHS. Under these circumstances, however, the first "s1" must also be copied, even though its right sibling (link) is not immediately modified. As a result of the 'X' s2 => s2 portion of the rule, the original subtree <'X' s1> 'A' s1 s1, which looks safe enough, may be effectively transformed into s1 'A' s1 s1, which requires copying the first s1 parameter as well. The philosophy adopted in the author's implementation is to copy all subtree parameters except the rightmost instance of a parameter in the RHS of a rule, unless its right sibling (link) is modified. If its right sibling is modified, it is unconditionally copied. A simpler strategy which might prove sufficient would involve

copying a subtree parameter when it is the entire tree to be returned by the application of a rule, or if the right sibling of a parameter is modified. This scheme still copies unnecessarily. For example, if the following one-rule T-grammar were implemented without copying, no ill effects would result:

'A' <'A' s1> s2 => <'A' s1 s2 >.

Another possibility for a minimal copying rule is the following one: Copy only those subtree parameters which occur more than once in the RHS of a rule.

It has not been proven which (if any) of the above copying strategies actually work in all cases without resulting in cycles in the intended "trees" or other ill effects. The entire copying question is a difficult one because of the variety of ways in which rules may interact with each other. It may turn out that the only good solution is to do a form of global analysis on the domain of trees to be considered and the set of transformation rules to be used on them. The difficulties of such an approach put its further discussion beyond the scope of this thesis.

Some Considerations in Applying T-grammars to Practical Problems

This section is intended primarily as a discussion of some practical considerations that are encountered when using a T-grammar based tree transformation system in actual

applications. One of the first considerations that is encountered is the nature of the interface between the external tree generator and the T-grammar tree transformer. In order for the tree transformer to correctly transform the trees produced by the external (user-supplied) tree generator, it must know how the symbol names the user tree generator creates will map into some internal representation of these node names. Unless all node names are kept in character-string format (a space-wasteful practice), a mapping function must be bound into the tree transformer. This binding may take place at any time prior to the actual application of a rule to a subtree. As is the case with most bindings, the earlier it is performed, the less overhead is incurred, and the later it is performed, the more flexibility is provided. Another important feature to include in a complete system is a provision for automatic generation of the binding information. This is easily done if the trees are generated by an automatically constructed tree generator such as a parsing algorithm generated automatically from a context-free grammar.

In the author's implementation, the symbol (node) names are bound to the internal representation when the user's T-grammar rules are translated into internal form. This makes the transformation-time binding trivial for the actual tree transformer. In this case, the mapping is the identity mapping, and need not incur any overhead while transforming trees. This binding is the earliest dynamic binding

possible. Since the translation of T-grammar rules into internal form is quite inexpensive, when the binding needs to be changed, the entire T-grammar can be easily (and cheaply) retranslated, thereby establishing the required new mappings.

The author's implementation is oriented towards using an LR(1) type of parser. More specifically, a parser using the SLR(1)/LALR(1)/LR(1) table generator described in [8,9] was used in this study. One of the outputs of this parsing table generator (and most others) is a symbol table. This symbol table is then input by the T-grammar rule translator which simply uses it to initialize its own symbol table, thereby effectively establishing the required binding. This is not a handicap, since ordinarily it is logically necessary to design the CF grammar for an application before designing the corresponding T-grammar for it. Note that the automatic determination of binding information is performed by reading in the automatically generated symbol table. Many other binding schemes would work as well as this one, but they will not be discussed further here.

Another problem which has to be considered when writing a translator for T-grammar rules is the problem of error detection. The detection of, and recovery from syntactic errors is not discussed here since a wealth of literature already exists on the subject [10,2,3]. Semantic errors are the other major class of errors. More specifically, this section will consider what types of errors can be detected

at T-grammar rule translation time, and what errors (if any), can only be detected when a specific rule is actually applied to a specific subtree. First, it is necessary to outline the basic types of semantic errors which might occur in syntactically correct rules. One of the most obvious errors that will be considered is an undefined rule parameter. That is, a parameter occurs in the RHS of a rule which is not defined in the LHS. One such rule is:

'A' t1 => <'B' t2>.

In this case, the value of the parameter "t2" is not defined in the LHS of the rule. The following rule is a slightly more subtle example of the same error.

'A'('B' | t1) => 'X' (1 t1 | 'Y').

In this rule, the value of the parameter t1 is well defined unless the actual subtree to be transformed is the tree <'A' 'B'>, in which case the value of the rule parameter t1 is used in the RHS without first having been given a value in the LHS. This condition can be detected when the rules are translated into internal form, if enough information is kept about the inherent nesting structure of the alternative-lists and parameter names. The problem is similar in many respects to the scope of identifiers problem for block-structured computer languages. The problem for T-grammars is more complex, however, since the parameters (objects being "declared") are defined in the LHS, of a rule, and used in the RHS. This would seem to imply a tree-structured symbol table rather than a stack-oriented symbol

table as is required by conventional block-structured languages.

Another type of semantic error which can occur is inconsistent alternative-list nestings. One example of a rule with an invalid alternative-list nesting is:

```
'A'('B' | (<'C' 'D') | 'E') t1 | 'F') =>
    (1 'X' | 'Y' 'Z') (2 'A1' | 'B1').
```

In this case, the second alternative-list in the RHS corresponds to the second alternative-list in the LHS. This alternative-list is used incorrectly in the RHS since it is not nested inside of the second subpattern of the first alternative list, as it is on the LHS. The problem here is similar to the previous problem concerning the conditionally undefined parameter values defined within parameter lists. Its detection can also be accomplished with the use of the same tree-structured symbol table required for detection of undefined parameters.

The other major semantic error occurs whenever a given alternative list has a different number of constituent subpatterns in the LHS than it does in the RHS of that same rule. An example of a rule containing this type of error is:

```
'A' ('B' | 'C' | 'D') => <'X' ('Y' | 'Z').
```

As can be easily seen, the alternative-list in the LHS has three sub-pattern alternatives, and the corresponding alternative-list in the RHS has only two. This rule has an undefined effect on the subtree <'A' 'D'>, since there is no

third alternative for the alternative-list in the RHS. This error is quite easy to detect and is easily accommodated within a simple symbol table structure.

In the author's implementation, however, most of these semantic error conditions are not detected at rule translation time, but are deferred until subtree-transformation time, when they are nearly all trivial to detect. This greatly simplifies the symbol table mechanism required to translate a given rule into internal form.

An additional ambiguity can arise if two rules match the same subtree. If such a situation should arise, the results would generally depend upon the order in which the rules were examined by the transformer. The existence of this condition is not usually obvious and may lead to undesirable results. It is also of some theoretical importance and is discussed at length in [15]. Detection of this condition should not be too difficult, and should probably result in a warning message rather than an error message. Such a warning message could prove useful in the detection of certain subtle errors or oversights in the design of a T-grammar.

Using T-grammars in Compilers

Compilers are a special class of programs with their own peculiar needs and problems. In a compiler, the external tree generator for the T-grammar transformer would generally be a syntax analyzer or parser for some context-

free language. As mentioned previously, the parser may operate in either a top-down (predictive) or bottom up fashion. A top-down parser can be made to construct its derivation tree from the bottom up by waiting until the RHS of the rule it is working on is completed before attaching the RHS subtrees to the node that corresponds to the LHS of the current rule in the CFG. After a subtree corresponding to an application of a rule in the CFG is constructed, the tree transformer module is invoked to determine whether the tree can be transformed according to any of the rules in the T-grammar. This process has the effect of constructing and transforming the tree from the bottom up, even though the parsing mechanism may be classified as being a "top-down" technique. The use of the T-grammar tree transformer with a bottom-up parser is essentially the same as with a top-down parser, except that the tree is built and the transformer is invoked after each reduction on a rule in the CFG is performed.

In a compiler, the (leaf) nodes of the syntax tree usually have information fields which are generally initialized by the lexical analyzer (scanner). In general, the values and manipulations of these application-dependent information fields are outside the realm of transformational grammars. Such an information field may contain, for example, a pointer to the symbol table to indicate which particular identifier is being referred to by the terminal symbol IDENTIFIER. Since these information fields are

usually necessary and present in the syntax tree, it may be desirable to manipulate them to some degree in a T-grammar. Towards this end, two primitive facilities are proposed: rule "predicates" and rule "actions". A rule predicate is simply a procedure which determines whether a particular subtree (and its information fields) meet some computable pre-condition. In other words, in order for a subtree transformation to take place, the given subtree must match both the LHS pattern and satisfy the specified pre-condition. If a subtree matches the LHS of a rule, and the predicate procedure returns true, then the subtree is replaced by a subtree constructed according to the RHS as before. It should be emphasized that these predicate procedures are written by the user in some conventional programming language which is callable by the T-grammar tree transformer. As such, these procedures may examine the information fields in any manner desired. One predicate which is invaluable in algebraic expression simplification problems is one which returns true if the values of the two rule parameters are identical. This would allow transformation rules to be written which could transform expressions of the form (A/A) into the constant 1, and expressions of the form (A*B) + (A*C) into expressions of the form A * (B+C).

The second proposed facility is a mechanism for performing some arbitrary action after transforming a subtree. These "actions" are patterned after the semantic

actions used in PL (Production Language) translators [10]. These action procedures may examine or modify any of the fields of any of the nodes in a subtree. Such actions might include: entering an identifier in a symbol table, copying the information field from one node to another, inserting a node in a subtree to indicate the type of an identifier (INTegral, REAL, CHARacter,etc.), or add two (numeric) information fields together. These two facilities act as "escape" mechanisms which allow the specification of transformations based upon the information fields, and operations upon the information fields of the tree nodes. Such operations are impossible without such a facility, since the information fields are outside the realm of the T-grammar formalism. Of course, the predicates need not depend solely upon the contents of the information fields, nor do the action procedures need to operate only upon these fields. Since these routines are coded in a conventional programming language, they may examine and/or modify global variables at will.

Once the decision is made to include the action and/or predicate extensions, it becomes necessary to decide the types and numbers of parameters these routines should be supplied. In the case of predicates, it would seem to be sufficient to pass them only the values of the rule's subtree parameters, since they are basically the only unknown quanities in the rules. In the case of action procedures, it may be desirable for the procedure to modify

the entire result tree. As a result, it seems desirable to pass the entire result subtree as a parameter to an action procedure. Additionally, it may be desirable to pass the values of the subtree parameters to the rules to an action procedure. These two mechanisms serve to provide a necessary communications link between the T-grammar transformer and the remainder of the compiler.

CHAPTER IV

CONCLUSIONS

This thesis consists of two major parts. The first part presents the concepts and applications of transformational grammars in an easily accessible manner. This part also clarified some semantic ambiguities which had not been discussed previously in the literature. Additionally, a novel type of subtree parameter is defined which simplifies certain implementation problems. The resulting T-grammar system has also been shown to be capable of generating any recursively enumerable set.

The second part of this thesis presents a discussion of some implementation considerations that were encountered when designing, implementing, and using the experimental T-grammar transformation system. Very little of this material has been presented in the literature previously. Several extensions to the experimental T-grammar system have also been suggested.

In conclusion, T-grammars comprise a powerful tree manipulation system which should prove useful in compiler writing, and other fields in computer science. Moreover, the experimental T-grammar implementation has shown that a practical T-grammar tree transformer system need not be difficult to implement.

A SELECTED BIBLIOGRAPHY

[1] Aho, A. and Ullman, J. D. The Theory of Parsing, Translation, and Compiling. Englewood Cliffs: Prentice-Hall, 1972.

[2] Aho, A. and Ullman, J. D. Principles of Compiler Design. Reading, Mass.: Addison-Wesley, 1977.

[3] Boullier, P. "Automatic Error Recovery for LR-parsers." in Fifth Annual III Conference on the Implementation and Design of Algorithmic Languages 5(1977), 349-361.

[4] Chomsky, N. Syntactic Structures. The Hague: Mouton, 1977.

[5] Cleaveland C. and Uzgalis, R. Grammars for Programming Languages New York: Elsevier North-Holland, 1977.

[6] De Remer, F. in Compiler Construction: An Advanced Course. New York: Springer-Verlag, 1977, 121-145.

[7] Ginsburg, S. and Partee, B. "A Mathematical Model of Transformational Grammars." Information and Control 15,4(1969), 297-334.

[8] Gray, J. "Implementation of a SLR(1) Parsing Algorithm." (Unpublished M.S. thesis; Stillwater, Oklahoma: Oklahoma State University), 1973.

[9] Gray, J. "Implementation of a LALR(1) Parsing Algorithm." (Addendum to [8]) (Unpublished Computer Center technical note; Stillwater, Oklahoma: Oklahoma State University), 1976.

[10] Gries, D. Compiler Construction for Digital Computers. New York: John Wiley and Sons, 1971.

[11] IBM. PL/I Checkout and Optimizing Compilers, Language Reference Manual. White Plains, N.Y.: IBM, 1976.

[12] Kleene, S. "Representaticn of Events in Nerve-sets", in Automata Studies, Princeton: Princeton University Press, 1956.

[13] Pagan, F. "On Interpreter Oriented Definiticns of Programming Languages." Computer J. 19,2(1976), 151-155.

[14] Robertson, A., Hedrick, G. E., and Goto M. "Grammars for ALGCL 68 Fcrmat Denotations and the Transput Facilities cf tle OSU ALGOL 68 Compiler." in Fifth Annual III Conference on the Implementation and Design cf Algcrithmic Languages 5(1977), 222-252.

[15] Rosen, B. "Tree Manipulating Systems and Church-Rosser Theorems." JACM 20,1(1973), 160-187.

[16] Rounds, W. "Mappings and Grammars on Trees." Mathematical Systems Theory 4,3(1971), 257-287.

APPENDIX A

A CONTEXT-FREE GRAMMAR TO GENERATE T-GRAMMAR RULES

1. GOAL : "?", TGRAMMAR, "?".

2. T-GRAMMAR: RULE;
   T-GRAMMAR, RULE.

4. RULE: LHS, =>, RES, PERIOD-SYMBOL.

5. LHS: L-TREE;
   NODENAME.

7. L-TREE: NODENAME, LSUBTREE-LIST;
   NODENAME, LSUBTREE-LIST, S-PARAMETER;
   NODENAME, S-PARAMETER.

10. LSUBTREE-LIST: LSUBTREE;
    LSUBTREE-LIST, LSUBTREE.

12. LSUBTREE: NODENAME;
    <, L-TREE, >;
    LCHOICE;
    T-PARAMETER.

16. LCHOICE: (, LALTERNATIVE-SEQ,).

17. LALTERNATIVE-SEQ: LSUBTREE-S-PARM;
    LALTERNATIVE-SEQ, |, LSUBTREE-S-PARM.

19. LSUBTREE-S-PARM: LSUBTREE;
    S-PARAMETER.

21. RHS: RSUBTREE-LIST.

22. RSUBTREE-LIST: RSUBTREE;
    RSUBTREE-LIST, RSUBTREE.

24. RSUBTREE: NODENAME;
    <, NODENAME, RUSBTREE-LIST, >;
    RCHOICE;

```
        T-PARAMETER;
        S-PARAMETER.

29.  RCHOICE: (, INTEGER, RALTERNATIVE-SEQ, ).

30.  RALTERNATIVE-SEQ: RSUBTREE;
        RALTERNATIVE-SEQ, |, RSUBTREE.

32.  T-PARAMETER: LETTER-T-SYMBOL, INTEGER.

33.  S-PARAMETER: LETTER-S-SYMBOL, INTEGER.

34.  INTEGER: DIGIT;
        INTEGER, DIGIT.

36. NODENAME: QUOTE-SYMBOL, CHAR-SEQ, QUOTE-SYMBOL.

37.  CHAR-SEQ: ANY-CHAR;
        CHAR-SEQ, ANY-CHAR.
```

APPENDIX B

A T-GRAMMAR TRANSLATOR FOR

ALGOL 68 FORMAT DENOTATIONS

A Context-Free Grammar For ALGOL 68 Format Denotations

```
 1.  GOAL : "?", PICTURELIST, "?".

 2.  PICTURELIST: PICCII;
        PICTURELIST, COMMA, PICCLL.

 4.  PICCLL: PICTURE;
        COLLECTION;
        INSERTION, PICTURE;
        INSERTION, COLLECTION;
        INSERTION.

 9.  INSERTION: ALITS;
        INSERTION, ALITS.

11.  COLLECTION: LPAREN, PICTURELIST, RPAREN;
        REPLICATOR, LPAREN, PICTURELIST, RPAREN;
        COLLECTION, AIITS.

14.  PICTURE: INTEGER;
        REAL;
        BITS;
        COMPL;
        STRING;
        OTHERPICINS.

20.  OTHERPICINS: OTHERPIC;
        OTHERPICINS, AIITS.

22.  OTHERPIC: BOOL;
        BOOLCH;
        GPIC;
        FPIC;
```

```
        INTCH.

27.  INTEGER: INITIAL-ZEROES;
        INT-FRAME.

29.  INITIAL-ZEROES: ZZ;
        INITIAL-ZEROES, ZZ;
        INITIAL-ZEROES, ALITS.

32.  INITIAL-SIGN: SIGN;
        INITIAL-SIGN, ALITS.

34.  SIGN: CHPLUS;
        CHMINUS;
        SCHPLUS;
        SCHMINUS.

38.  INT-FRAME: D-OR-SD-OR-SZ;
        INT-FRAME, D-OR-SD-OR-SZ;
        INT-FRAME, ZZ;
        INITIAL-SIGN, D-OR-SD-OR-SZ;
        INITIAL-SIGN, ZZ;
        INITIAL-ZEROES, CHPLUS;
        INITIAL-ZEROES, CHMINUS;
        INITIAL-ZEROES, D-OR-SD-OR-SZ;
        INT-FRAME, ALITS.

47.  ZZ: LZ;
        REPLICATOR, LZ.

49.  D-OR-SD-OR-SZ: D-SD-SZ;
        REPLICATOR, D-SD-SZ.

51.  REAL: FREAL;
        EREAL.

53.  FREAL: INTEGER, POINT;
        INTEGER, POINT, ZDSEQ;
        POINT, ZDSEQ.

56.  ZDSEQ: ZZ;
        D-OR-SD-OR-SZ;
        ZDSEQ, ZZ;
        ZDSEQ, D-OR-SD-OR-SZ;
        ZDSEQ, ALITS.

61.  POINT: DOT-SYMB;
        POINT, ALITS.

63.  EREAL: FREAL, EE, INTEGER.

64.  EE: E-SYMB;
```

```
        EE, ALITS.

66.  COMPL: REAL, II, REAL.

67.  II: I-SYMB;
        II, ALITS.

69.  BITS: RADIX, ZDSEQ.

70.  RADIX: INT, LR;
        RADIX, ALITS.

72.  STRING: AA;
        STRING, AA;
        STRING, ALITS.

75.  AA: A-SYMB;
        REPLICATOR, A-SYMB.

77.  BOOL: LB;
        LSB.

79.  BOOLCH: LB, LPAREN, LITS, COMMA, LITS, RPAREN.

80.  INTCH: LC, LPAREN, STRINGLIST, RPAREN.

81.  STRINGLIST: LITS;
        STRINGLIST, COMMA, LITS.

83.  FPIC: LF, FORMATUNIT.

84.  ALITS: ALIGN;
        LITS.

86.  LITS: LITERAL;
        REPLICATOR, LITERAL.

88.  ALIGN: ALIGNMENT;
        REPLICATOR, ALIGNMENT.

90.  REPLICATOR: INT;
        LN, INTUNIT.
```

A T-grammar for Translating ALGOL 68
Format Denotations

```
'PICTURELIST' < 'PICTURELIST' s1> =>
        <'PICTURELIST' s1>.
```

```
'PICTURELIST' <'PICTURELIST' s1> 'COMMA' s2 =>
        <'PICTURELIST' s1 s2>.

'PICOLL' t1 => t1.

'INSERTION' <'INSERTION' s1> s2=> <'INSERTION' s1 s2>.

'COLLECTION' <'COLLECTION' s1> s2 =>
        <'COLLECTION' s1 s2>.


'COLLECTION' <'REPLICATOR' t1> 'LPAREN' t2 'RPAREN' =>
        <'REP-GROUP' s1 t2>.

'COLLECTION' 'LPAREN' t1 'RPAREN' => t1.

'PICTURE' s1 => s1.

'OTHERPICINS' <'OTHERPICINS' s1> s2 =>
        <'OTHERPICINS' s1 s2>.

'OTHERPIC' s1 => s1.

'INT-FRAME' s1 => s1.

'INITIAL-ZEROES' s1 => s1.

'INITIAL-SIGN' s1  => s1.

'SIGN' s1 => s1.

'PLUSMINUS' s1 => s1.

'PICOLL' <'INSERTION' s1>
        (<'INTEGER' s2> | <'REAL' s3> | <'BITS' s4>
        |<'COMPL' s5> | <'STRING' s6> | <'BOOL' s7>
        |<'BOOLCH' s8> | <'GPIC' s9> | <'FPIC' s10>
        |<'INTCH' s11> | <'COLLECTION' s12>   )
 =>
        (1  <'INTEGER' s1 s2> | <'REAL' s1 s3>
             | <'BITS' s1 s4>
        |<'COMPL' s1 s5> | <'STRING' s1 s6>
              | <'BOOL' s1 s7>
        |<'BOOLCH' s1 s8> | <'GPIC' s1 s9>
              |<'FPIC' s1 s10>
        |<'INTCH' s1 s11> | <'COLLECTION' s1 s12>  ).

'ZDSEQ' s1 => s1.

'EREAL' <'FREAL'  s1> s2 => <'EREAL' s1 s2>.

'EE' s1 => s1.
```

```
'REAL' s1 => s1.

'POINT' s1 => s1.

'EE' s1 => s1.

'II' s1 => s1.

'RADIX' <'RADIX' s1> s2 => <'RADIX' s1 s2>.

'STRING' <'STRING' s1> s2  => <'STRING' s1 s2>.

'BOOLCH' 'LB' 'LPAREN' t1 'COMMA' t2 'RPAREN'  =>
        <'BOOLCH' t1 t2>.

'INTCH' 'LC' 'LPAREN', t1, 'RPAREN'  =>
        <'INTCH' t1>.

'STRINGLIST' <'STRINGLIST' s1> s2 =>
         <'STRINGLIST' s1 s2>.

'ALITS' s1 => s1.
```

APPENDIX C

A T-GRAMMAR TURING MACHINE

An important property pcssessed by the T-grammar formalism is that it is able to generate any recursively enumerable set. This property essentially implies that a T-grammar can solve any problem which can be solved by any other computer program. This property can be established if a T-grammar system can be shcwn to be capable of simulating an abstract computing device called a Turing machine. A Turing machine is a finite state device with an unbounded amount of memory in the form cf a tape which is divided up into squares each of which can hold a symbol. For any given state and (current) input symbcl, the machine performs the following sequence of actions: establish a new state, write a single symbol to the tape, and move the tape head one square to the left or right. Fcrmally a Turing machine may be defined as a 6-tuple (Q,G,Σ,delta,g0,b), where Q is the set of states, G is the set of permissible tape symbols, Σ is the set of input symbols (a subset of G), g0 is designated as the start state, b is an element of G-Σ which has been designated as the blank, delta is the finite state control function which maps elements of Q x G into elements of Q x G x D. In this definition, D≡{L,R}, representing the

direction of tape movement, Left or Right. A configuration of the Turing machine is a pair (q,a c b) where q is the current state, and a c b is the non-blank portion of the tape. "c" is the current symbol, "a" is the portion of the tape to the left of the current symbol, and "b" is the portion of the tape to the right of the current symbol. The next configuration of the machine is computed from delta(current state,current symbol) and consists of a triple (s',t',D) where s' is the new state after this action, t' is the tape symbol to be written onto the tape to replace the current symbol, and D is the direction of movement of the tape. These configuration properties are represented in the T-grammar Turing machine simulator by subtrees of a tree to be transformed. The elements of delta are each represented by a rule in the T-grammar. The control mechanism is represented by three T-grammar rules. A typical configuration of the T-grammar Turing machine is shown below:

```
<'TURING-MACHINE'
  <'ST-SYM'
    <'STATE' 'current state'>
    <'CURR-SYM' 'current tape symbol'> >
  <'TAPE'
    <'BEFORE' 'symbols before current symbol'>
    <'AFTER' 'symbols after current symbol'> > >.
```

In this model, the current state is represented by the single subtree of 'STATE' and the current tape symbol is represented by the single subtree of 'CURR-SYM'. The tape is represented by the two subtrees of the 'TAPE' node. The squares of the tape to the left of the current symbol are

represented by the subtrees of the 'BEFORE' node, and the subtrees to the right of the current symbol are represented by the subtrees of the 'AFTER' node. The control mechanism of the Turing machine is represented by the following three T-grammar rules:

```
'TURING-MACHINE' <'MOVE' t1 t2 ('L' | 'R')> =>
  (<'READ-NEXT' t1 t2 t3> | <'READ-PREV' t1 t2 t3>).

'READ-NEXT' t1 t2 <'TAPE' < 'BEFORE' s3>
  <'AFTER' (t4 s5 | t6)> =>
  <'TURING-MACHINE'
    <'ST-SYM'
      <'STATE t1>
      <'CURR-SYM' (t4 | t6)>
    <'TAPE'
      <'BEFORE' t2 s3>
      <'AFTER' (1 s3 | 'BLANK')> > >.

'READ-PREV' t1 t2 <'TAPE' <'BEFORE' (t3 s4 | t5)>
    <'AFTER' s6> > =>
  <'TURING-MACHINE'
    <'ST-SYM'
      <'STATE t1>
      <'CURR-SYM' (1 t3 | t5)>
    <'TAPE'
      <'BEFORE' (1 s4 | 'BLANK')>
      <'AFTER' t2 s6> > >.
```

In this T-grammar model, the elements of the next move function (delta) are each represented by a rule in the T-grammar. For each value of delta(q,A)=(p,B,D), a rule of the form shown below is constructed.

```
'ST-SYM' <'STATE' 'q'><'NEXT-SYM' 'A'> =>
        <'MOVE' 'p' 'B' 'D'>.
```

The application of a rule of this form forces the transformer to apply two of the three control mechanism rules to "write" the tape symbol, change the state, and "move" the tape. As an example, the following Turing

machine delta function counts the number of "1"'s on its input tape and writes the number of 1's MOD 3 onto the tape.

```
delta≡{(zerostate,'1'), (onestate,'BLANK',R)),
       ((onestate,'1'), (twostate,'BLANK',R)),
       ((twostate,'1'), (zerostate,'BLANK',R)),
       ((zerostate,'BLANK'), (halt,'ZERO',R)),
       ((onestate,'BLANK'), (halt,'ONE',R)),
       ((twostate,'BLANK'), (halt,'TWO',R)) }.
```

These function values are easily represented by the following set of T-grammar rules:

```
'ST-SYM' <'STATE' 'zerostate'> <'CURR-SYM' '1'> =>
  <'MOVE 'onestate' 'BLANK' 'R'>.

'ST-SYM' <'STATE' 'onestate'> <'CURR-SYM' '1'> =>
  <'MOVE 'twostate' 'BLANK' 'R'>.

'ST-SYM' <'STATE' 'twostate'> <'CURR-SYM' '1'> =>
  <'MOVE 'zerostate' 'BLANK' 'R'>.

'ST-SYM' <'STATE' 'zerostate'> <'CURR-SYM' 'BLANK'>  =>
  <'MOVE 'halt' 'ZERO' 'R'>.

'ST-SYM' <'STATE' 'onestate'> <'CURR-SYM' 'BLANK'> =>
  <'MOVE 'halt' 'ONE' 'R'>.

'ST-SYM' <'STATE' 'twostate'> <'CURR-SYM' 'BLANK'> =>
  <'MOVE 'halt' 'TWO' 'R'>.
```

This example illustrates how a particular Turing machine simulator is constructed from its mathematical definition. This example is easily extended to all Turing machines as indicated above. The definition of the tape symbol sets is done implicitly by the initialization of the tree and the definition of the delta function transformation rules.

VITA[2]

Alan Lynn Robertson

Candidate for the Degree of

Master of Science

Thesis: TRANSFORMATIONAL GRAMMARS: THEIR APPLICATIONS AND IMPLEMENTATION

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Lincoln, Nebraska, May 18, 1954, the son of Mr. and Mrs. D. Keith Robertson.

Education: Graduated from Clinton High School, Clinton, Oklahoma, in May, 1972; received the Bachelor of Science degree from Oklahoma State University in May, 1976, with a major in Electrical Engineering; completed requirements for the Master of Science Degree in May, 1978.

Professional Experience: Research Associate, Oklahoma State University, Department of Computing and Information Sciences, Stillwater, Oklahoma, August, 1976 to May, 1978; computer programmer, Continental Oil Company, Ponca City, Oklahoma, May, 1975 to August, 1975.