

## **General Disclaimer**

### **One or more of the Following Statements may affect this Document**

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.
- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.
- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.
- This document is paginated as submitted by the original source.
- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

A PROPOSED TRANSLATOR-WRITING-SYSTEM LANGUAGE

FRANKLIN L. DE REMER

CEP REPORT, VOLUME 3, No. 1

September 1970

FACILITY FORM 602

|                                                   |                         |
|---------------------------------------------------|-------------------------|
| <b>N71-19596</b><br>(ACCESSION NUMBER)            |                         |
| <b>43</b><br>(PAGES)                              | <b>63</b><br>(THRU)     |
| <b>CR-117124</b><br>(NASA CR OR TMX OR AD NUMBER) | <b>08</b><br>(CODE)     |
|                                                   | <b>08</b><br>(CATEGORY) |



The Computer Evolution Project  
Applied Sciences  
The University of California at  
Santa Cruz, California 95060

## A PROPOSED TRANSLATOR-WRITING-SYSTEM LANGUAGE

### ABSTRACT

We propose herein a language for an advanced translator writing system (TWS), a system for use by language designers, as opposed to compiler writers, which will accept as input a formal specification of a language and which will generate as output a translator for the specified language. Although the TWS language is designed primarily for specifying programming languages, and therefore compilers, it is general enough to be found useful in such areas as symbolic mathematics and natural language translation and interpretation.

The language is based on an integration of regular expressions, context-free grammars, cascaded transduction grammars, and tree attributes. It provides a single powerful and convenient method for specifying the lexicon, macros, and syntax of a language and the associated translations.

The language is an extension of an existing, well documented language, namely McKeeman's XPL, and it could be implemented as an extension of the XPL system. The practical, extended LR(k) techniques of the author for generating highly efficient parsers make the TWS language practicable. To facilitate post-syntactic language specification, a new data type similar to "tuples" and "plexes" must be added to XPL for the convenient representation of trees and control structures.

## TABLE OF CONTENTS

|                                                            |     |
|------------------------------------------------------------|-----|
| Title Page                                                 | i   |
| Abstract                                                   | ii  |
| Table of Contents                                          | iii |
| I. Introduction                                            | 1   |
| II. Long and Short Term Objectives                         | 2   |
| III. The Proposed TWS Language: Illustrated by Example     | 4   |
| A. PREFACE                                                 | 4   |
| B. THE COMPILER IMPLIED BY FIGURE 1                        | 9   |
| C. XPL_LEXICAL_ANALYSER                                    | 14  |
| D. XPL_PREPROCESSOR                                        | 20  |
| E. XPL_SYNTACTICAL_ANALYSER                                | 22  |
| F. FEATURES NOT ILLUSTRATED ABOVE                          | 24  |
| IV. The Proposed TWS Language: Formal Definition (partial) | 26  |
| V. Recent Results that make the TWS Feasible               | 26  |
| VI. Comparison with Previous TWS Efforts, Evaluation       | 33  |
| VII. Summary                                               | 36  |
| Appendix (Figure A1)                                       | 37  |
| Acknowledgments                                            | 38  |
| References                                                 | 39  |

## A PROPOSED TRANSLATOR-WRITING-SYSTEM LANGUAGE<sup>+</sup>

### I. Introduction

We propose herein a language for an advanced translator writing system (TWS), a system for use by language designers, as opposed to compiler writers, which will accept as input a formal specification of a language and which will generate as output a translator for the specified language.

The language represents a significant step forward in the TWS area. It provides a single, powerful and convenient method for specifying the lexicon, macros, and syntax of a language and the associated translations. This method is based on an integration of regular expressions, context-free grammars, cascaded transduction grammars (L&S 68), and tree attributes (Knu 66).

The convenient methods for specifying grammars and translations are practicable due to the existence of techniques for constructing highly efficient parsers for a large class of grammars -- the left-to-right translatable context-free grammars with regular expressions as right parts of productions (DeR 70c). These techniques are founded in the LR(k) techniques of Knuth (Knu 65) and in automata theory (DeR 69).

The TWS language includes general data structures, similar to the "tuples" of PAL (Eva 69) and the "plexes" of AED (Ros 67),

---

<sup>+</sup>Work reported was supported in part by NASA under grant number NGR05-061-005.

for representing trees and control structures. Several built-in functions provide for the convenient manipulation of these structures. Construction of most of the structures are specified in the translation parts of rules based on Bachus Naur Form (BNF)-like productions.

The underlying TWS could be implemented as an extention of the best documented TWS currently in existence, namely McKeeman's XPL system; witness (MHR 70) and SHARE PID 03.2.015.

## II. Long and Short Term Objectives

Before describing the proposed TWS language, we discuss our long and short term objectives in the TWS area.

Our ultimate goal, the realization of which awaits several important developments in Computer Science, is a system that might better be called a language implementing system (LIS), than a TWS. We envision the user of this LIS supplying as input to the system a formal specification of a language L, some statistics regarding the expected use of L and an underlying computing system, and an amount of money that the LIS user is willing to spend to implement L; the output of the LIS would be specifications of both an appropriate computing machine (system) M and a translator (compiler) from L to the machine language of M. More generally, we envision the LIS accepting the specifications of several languages and generating the specifications of an appropriate host computing machine for all the languages and a translator for each of them.

Of course the implementation of an acceptable LIS will be impossible until three areas are better understood and better formalized: programming language design and specification, computing machine design and specification, and artificial intelligence techniques related to the optimal design of machines and "intelligent" translators. The first of these is currently the area of primary interest to the author. We have broadened our scope and discussed a system that involves all three areas to emphasize that our ultimate goal is a system for use by language designers, not compiler writers. In the case of an LIS, the machine is designed only after the languages have been specified, so the methods of language specification could hardly be machine dependent.

Our goal for the proposed language, however, is more in line with the conventional notion of a TWS. We retain the idea that our system should be for use by language designers, but we assume a pre-existing computing machine. Thus, we are left with the problem of designing a language-description language that is as machine-independent as possible, but whose programs (language specifications) can be mapped into translators for a specific computer. We believe that we can now attain this goal and make a stride forward in this area by developing a tool that will spawn further progress toward an LIS.

We now proceed to describe our proposed language. Afterwards, we compare it with existing TWS's and assess its proximity to our restricted goal.

### III. The Proposed TWS Language: Illustrated by Example

#### PREFACE

Figure 1 illustrates the use of our language to specify an existing, practical programming language. The language is XPL, an extended subset of PL/I and the very language that we propose to extend to form our TWS language. Thus, when we have specified in a similar manner below the extensions that we propose, we will have given a formal description of the lexicon, syntax, and translation to "abstract syntax" (Bee 70, Lan 66, McCa 66) of our TWS language, TWS/1. The bootstrapping technique used here to describe our language parallels the technique that would be used to implement it; i.e., we would bootstrap our way up from the existing TWS.

Syntactically, the primary change to the XPL language is the addition of the category <translator definition> and its dependents, the definitions of which are given in the following section. Of course, corresponding semantic additions must be made, along with the addition of a new data type for representing trees and control structures.

The reader will note during the discussion below that most of the methods of compiler writing have been dispensed with (or disguised) in favor of methods of language specification. The machine dependent and compiler-related techniques that must be retained are hidden in procedures that are referred to but not specified here. Those procedures are assumed to be either predefined or written in existing XPL, except for the use of the new data type to be added to the language. To facilitate the

```

/* SYNTAX:
/* (IN HNF- AND REGULAR EXPRESSION-LIKE FORM)

XPL_LEXICAL_ANALYSER:
TRANSLATOR ( CARD_READER );

/* FIRST, WE MAKE SOME PRELIMINARY DEFINITIONS. */

BINARY_DIGIT = '0' | '1';
QUARTAL_DIGIT = '0' | '1' | '2' | '3';
OCTAL_DIGIT = QUARTAL_DIGIT | '4' | '5' | '6' | '7';
DIGIT = OCTAL_DIGIT | '8' | '9';
HEX_DIGIT = DIGIT | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';

LETTER = 'A' | 'R' | ... | 'Z' | '$' | '#' | '_' | '.';

SYM_LESS_QUOTE = ERCDIC - '...';

/* NEXT, WE DEFINE THE BASIC "TEXTUAL ELEMENTS" OR "WORDS" OF XPL. */

INTEGER = DIGIT+
BIT_STRING = '"' ( HEX_INTEGER | BIT_GROUP ) BIT_GROUP# '"' ;
    HEX_INTEGER = HEX_DIGIT+
        BIT_GROUP = ('1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F');
NUMBER = INTEGER | BIT_STRING

STRING = '...' ( SYM_LESS_QUOTE | SINGLE_QUOTE )# '...' => *
SINGLE_QUOTE = '...' '...'

RESERVED_WORD = 'I' | 'F' | 'T' | 'H' | 'E' | 'N' | ... | 'R' | 'Y' => * IF | THEN | ... | 'RY' ;

IDENTIFIER = LETTER ( LETTER | DIGIT )* - RESERVED_WORDS => * $NAME_LIST( RIGHT_PART IDENTIFIER ) ;

OPERATOR = '+' | '-' | ... | ':' => * RIGHT_PART; /* ALL SYMBOLS USED AS OPERATORS. */

/* NOW WE DEFINE IGNOREABLE SEPARATORS (I.E., "SPACES").
NULL TRANSLATIONS FOR ALL BUT ONE OF THE FOLLOWING. */

SPACE = ' ' | COMMENT;
COMMENT = '/' '*' ( SYM_LESS_AST | '*' SYM_LESS_SLASH )# '*'? '*' '/';
SYM_LESS_DOLLAR = ERCDIC - '$';
DOLLAR = '$' ERCDIC
SYM_LESS_AST = SYM_LESS_DOLLAR | DOLLAR - '*';
SYM_LESS_SLASH = SYM_LESS_DOLLAR | DOLLAR - '/';

/*-----*/
/* FIGURE 1. AN EXAMPLE OF A PROGRAM WRITTEN IN THE LANGUAGE TWS/1. THE PROCEDURES (TRANSLATIONS) DESCRIBE THE LOGIC OF ANALYSIS.
/* TACTICAL ANALYSERS AND THE PREPROCESSOR FOR THE LANGUAGE XPL, THE BASE FOR TWS/1. (PAGE ONE OF TWO PAGES)
/*-----*/

```

```

/* FINALLY, WE DEFINE AN XPL TEXT: I.E., WE SPECIFY WHERE SPACES ARE ALLOWED AND REQUIRED. */

```

```

XPL_TEXT = SPACE* WORD_STRING? SPACE* ( OP_STRING SPACE* WORD_STRING )* SPACE* OP_STRING? SPACE* :
WORD = NUMBER | STRING | RESERVED_WORD | IDENTIFIER:
WORD_STRING = WORD ( SPACE* WORD )* :
OP_STRING = OPERATOR ( SPACE* OPERATOR )* :

```

```

END XPL_LEXICAL_ANALYSER:

```

```

/*-----
/*
/* ALTERNATE DEFINITIONS FOR "OPERATOR" AND "OP_STRING".
/*
/* OPERATOR = OP_LESS_2 | TWO_OPS:
/* TWO_OPS = '|' | '*'
/* OP_LESS_2 = LCNG_OP | '+' | '-' | '.' | ':'
/* LONG_OP = '*' | '*' | '|' | ':'
/* OP_STRING = OP_STR_OP_LESS_2 | OP_STR_TWO_OPS :
/* OP_STR_OP_LESS_2 = OP_LESS_2 | OP_STRING SPACE* OP_LESS_2 :
/* OP_STR_TWO_OPS = TWO_OPS | OP_STR_OP_LESS_2 SPACE* TWO_OPS | OP_STR_TWO_OPS SPACE* TWO_OPS :
/*
/*-----

```

```

XPL_PREPROCESSOR:

```

```

TRANSLATOR ( XPL_LEXICAL_ANALYSER ) :

```

```

PROGRAM = STRING
STRING = GEN_SYMBOL+
GEN_SYMBOL = GROUP | BLOCK | LITERAL_DEC | BASIC_SYMBOL =>
GROUP = 'CO' ':' STRING 'END'
BLOCK = HEAD STRING 'END'
HEAD = 'BEGIN' | 'PROCEDURE' | 'DO' NON_SEMICOLON
NON_SEMICOLON = 'WHILE' | 'CASE' | 'IDENTIFIER'
LITERAL_DEC = 'DECLARE' LITERAL ( ',' LITERAL )* ':'
LITERAL = 'IDENTIFIER' 'LITERALLY' 'STRING'
BASIC_SYMBOL = 'NUMBER' | 'STRING' | 'IF' | 'THEN'
| '.' | 'BY' | 'IDENTIFIER' | '|' | '*'
| '+' | '-' | '.' | ':' | '*' | '|'
- ( 'BEGIN' | 'PROCEDURE' | 'DO' | 'END' | 'LITERALLY' )
=> RIGHT_PART:
/* ALL NAME NAMES OUGHT BY THE XPL_LEXICAL_
/* ANALYSER EXCEPT THOSE SPECIFICALLY
/* SUBSTITUTED.

```

```

END XPL_PREPROCESSOR:

```

```
XPI_SYNTACTICAL_ANALYSER:
TRANSLATOR ( XPL_PREPROCESSOR );
```

```
/* AT THE OUTERMOST LEVEL AN XPL PROGRAM IS A STATEMENT LIST. */
```

```
XPL_PROG = ST_LIST
ST_LIST = ST+
ST = IF_ST | RA_ST
=> *XPL_SYNTACTICAL_ANALYSER( ST_LIST );
=> *RURALF_LABELS( ST+ ';;' );
=> RIGHT_PART;
```

```
/* A CONDITIONAL (IF) STATEMENT. */
```

```
IF_ST = 'IF' L_EXP 'THEN' ST
| 'IF' L_EXP 'THEN' RA_ST 'ELSE' ST
| 'IDENTIFIER' ':' IF_ST
=> *RURALF_LABELS( L_EXP ST 'NO_OP' 'IF' );
=> *RURALF_LABELS( L_EXP RA_ST ST 'IF' );
=> 'IDENTIFIER' IF_ST ';;';
```

```
/* A BASIC STATEMENT. */
```

```
RA_ST = 'IDENTIFIER' ':' RA_ST
| VAR ( ':' VAR ) * '=' L_EXP ';;'
| 'GOTO' VAR ';;'
| 'GO' 'TO' VAR ';;'
| 'CALL' VAR ';;'
| 'RETURN' ';;'
| 'RETURN' L_EXP ';;'
| 'DECLARE' DE ( ':' DE ) * ';;'
| 'PROCEDURE_DEF' ';;'
| 'BLOCK' ';;'
| 'GROUP' ';;'
| ';;'
=> 'IDENTIFIER' RA_ST ';;' ;
=> VAR VAR* L_EXP 'ASSIGN' ;
=> VAR 'GOTO' ;
=> VAR 'GO' ;
=> VAR 'CALL' ;
=> 'RETURN' ;
=> L_EXP 'RETURN_VALUE' ;
=> DE DE* 'DECLARE' ;
=> 'PROCEDURE_DEF';
=> 'BLOCK';
=> 'GROUP';
=> 'NO_OP' ;
```

```
/* SOME AUXILIARY DEFINITIONS NEEDED ABOVE. */
```

```
DE = ( 'IDENTIFIER' | ID_LIST ) ( '(' 'NUMBER' ')' ) ? TYPE INITIAL_LIST?
=> ( 'IDENTIFIER' | ID_LIST ) 'NUMBER' ? TYPE INITIAL_LIST ';;'
ID_LIST = '(' 'IDENTIFIER' ( ':' 'IDENTIFIER' ) * ')' => 'IDENTIFIER' 'IDENTIFIER' * ';;' ;
INITIAL_LIST = ( 'INITIAL' '(' 'CONSTANT' ( ':' 'CONSTANT' ) * ')' ) +
=> ( 'CONSTANT' 'CONSTANT' * ':' ) + 'INITIAL_VALUES' ;
TYPE = 'FIXED' | 'CHARACTER' | 'LABEL'
| 'BIT' '(' 'NUMBER' ')' ;
=> RIGHT_PART ;
=> 'NUMBER' 'BIT' ;
```

```

PROCEDURE = PROC_HEAD ST_LIST ENDING
PROC_HEAD = IDENTIFIER ' ' PROCEDURE ID_LIST TYPE? ' '
=> IDENTIFIER ID_LIST TYPE? PROCEDURE :
=> ST_LIST ENDING BLOCK :

BLOCK = 'BEGIN' ' ' ST_LIST ENDING

GROUP = GROUP_HEAD ST_LIST ENDING
GROUP_HEAD = 'DO' VAR ' ' L_EXP ' ' L_EXP ' ' L_EXP? ' '
=> VAR L_EXP L_EXP L_EXP? ' ' :
=> L_EXP 'WHILE' :
=> L_EXP 'CASE' :
=> 'DO' :
=> IDENTIFIER ' ' ENDING :
=> IDENTIFIER? 'END' :

/* NOW THE EXPRESSIONS. */

L_EXP = L_FACT | L_FACT ( ' ' L_FACT ) +
L_FACT = L_SFC | L_SFC ( ' ' L_SFC ) +
L_SFC = L_PRI | ' ' L_PRI
L_PRI = S_EXP | S_EXP RLN S_EXP
RLN = '>' | '<' | '=' | '>' | '<' | '=' | '<' | '>'
| ' ' | '=' | '>' | '<' | ' ' | '<' :

S_EXP = A_EXP | A_EXP ( ' ' A_EXP ) +

A_EXP = ( ' ' | ' ' ) A_TERM
| ( ' ' | ' ' ) A_TERM ( ' ' | ' ' ) A_TERM +
A_TERM = A_PRI | A_TERM ( ' ' | ' ' ) A_PRI
A_PRI = VAR | CONSTANT | ' ' L_EXP ' '

VAR = IDENTIFIER | IDENTIFIER ' ' SURS_LIST ' '
SURS_LIST = L_EXP ( ' ' L_EXP ) *
CONSTANT = 'STRING' | 'NUMBER'

```

END XPL\_SYNTACTICAL\_ANALYSER:

```

/*-----*/
/* FIGURE 1. CONTINUED. (PAGE FOUR OF FOUR PAGES.) */
/*-----*/

```

+Actually XPL has no BLOCKs. We have added them for illustrative purposes.

writing of such procedures XPL provides such features as compile-time facilities (macros), list strings with operators and built-in functions for their manipulation, and a method of executing arbitrary machine-language instructions. Aside from such special features, XPL is best described as a powerful, higher-level programming language, a necessary component of any good, contemporary TWS.

Since the new data type to be added to XPL is not novel, being essentially the same as the "tuple" of PAL and the "plex" of AED, we will not discuss it here except to note (a) that it is used to represent the trees generated as output by the translators exemplified below, and (b) that our main problems in this area will be to decide which related functions to provide as predefined and how to represent these structures internally in an optimal way.

#### THE COMPILER IMPLIED BY FIGURE 1

In the sequel we describe Figure 1 primarily as the specification of the lexicon, macros, syntax, and translation to "abstract syntax" of XPL. Nevertheless, the primary reason for describing XPL in the language TWS/1 would normally be to define an XPL compiler. We first describe the organization of the implied compiler, which is presumably named XPL\_COMPILER at some outer level not shown in Figure 1, and then we discuss specific constructs in TWS/1.

Conceptually, CARD\_READER may be thought of as a procedure that reads cards column-by-column and sends its results to the XPL\_LEXICAL\_ANALYSER; this is indicated by the header for the latter in Figure 1. The XPL\_LEXICAL\_ANALYSER then effects a translation and sends its results to the XPL\_PREPROCESSOR, which in turn sends its output to the XPL\_SYNTACTICAL\_ANALYSER. The output (abstract syntax tree) from the latter is received by an XPL\_SYNTHESIZER, which translates the tree into object code.

The appropriate communications links, such as queues, deques, or buffers, are established between these translators automatically by the system. Furthermore, the system makes the decision whether to effect the various translations simultaneously or sequentially, on the basis of space and time considerations, unless it is specifically directed with regard to such decisions by the language designer (compiler writer, in this capacity).

It is to be emphasized that after being processed by the CARD\_READER the program being compiled is represented by terminal and nonterminal tree nodes, at least through the abstract syntax stage. Thus, the CARD\_READER reads (from a file containing card or line images) a sequence of characters and converts it into a sequence of terminal nodes whose names are the characters of the original sequence. Each subsequent translator reads the nodes, some terminal and some nonterminal, that are output as translations by the previous stage. The process is

illustrated graphically in Figure 2<sup>†</sup>.

Our grammars, then, which are used to define translators, should be viewed in an unusual way. The terminal symbols in these grammars should be regarded as the names of nodes that may or may not have substructure (descendants). Whether or not these nodes have substructure is, in general, defined by another grammar. Thus, we have the notion of a cascade of grammars as a device for specifying languages at various "levels" or in various "components". Correspondingly, each of our compilers is a cascade of translators.

Each grammar should be regarded as generating a set of strings (sequences) of nodes and as associating with each string a translation. The translation is itself a sequence of nodes, each of which is usually constructed from (or over) the generated string.

The first few translators in our compilers, then, start with a sequence of terminal nodes and level-by-level build over these nodes an abstract syntax tree; i.e., the tree associated

---

<sup>†</sup>The above is a good conceptual description of the compiler, but in practice the word CARD\_READER will be a predefined identifier that will cause the XPL\_LEXICAL\_ANALYSER to be an unusual translator which reads not nodes but characters. In other words, the system will automatically merge the CARD\_READER and XPL\_LEXICAL\_ANALYSER for the sake of efficiency.

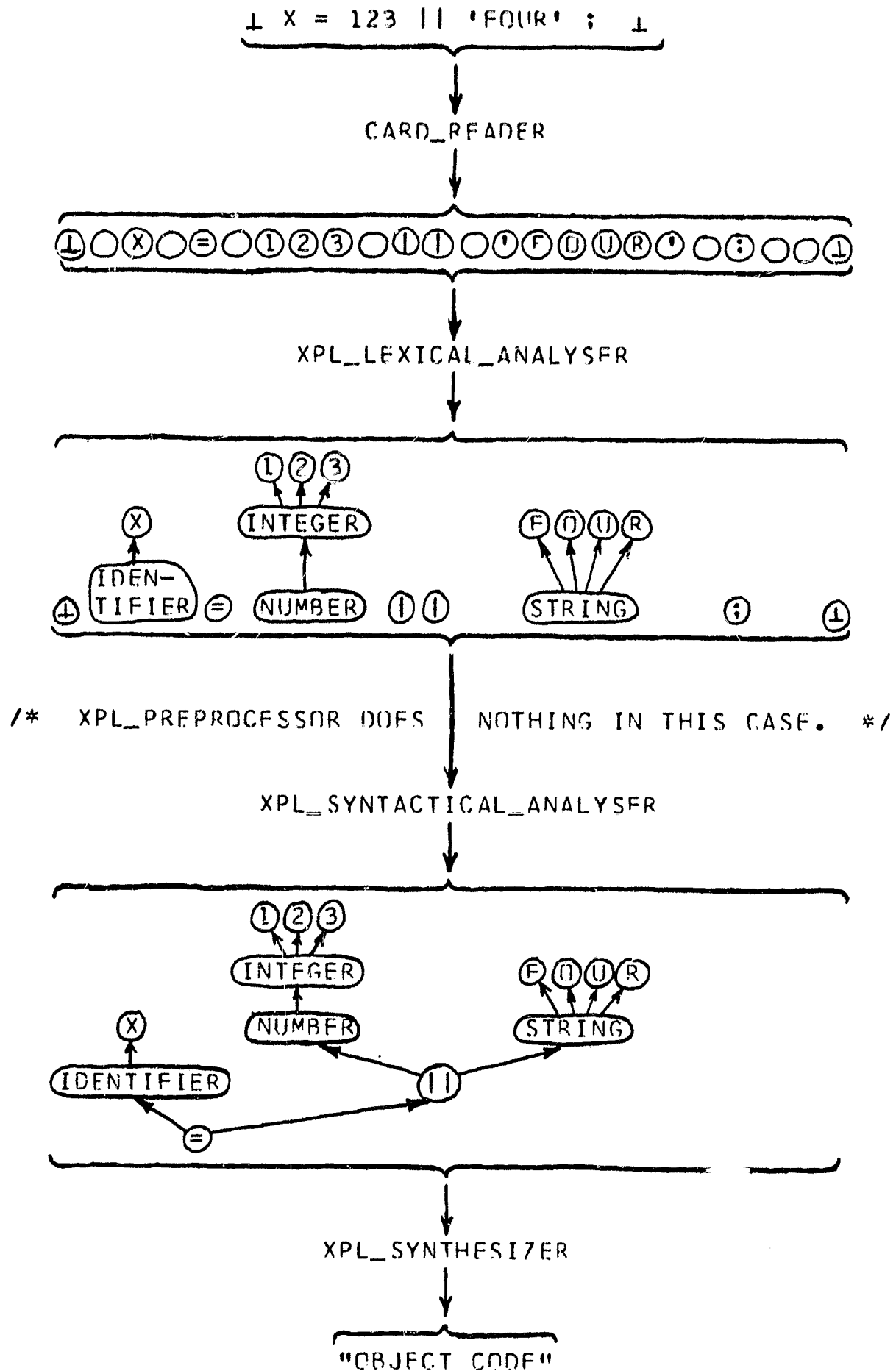


FIGURE 2. PICTORIAL DESCRIPTION OF THE COMPILING PROCESS. ACTUALLY, THE FIRST TRANSFORMATION INDICATED ABOVE IS INCORPORATED INTO THE XPL\_LEXICAL\_ANALYSER.

with the original sequence by the cascade of grammars. Of course, some of the original nodes may not be involved in the final tree; e.g., spaces; others will have been converted from terminal to nonterminal nodes; e.g., operators; and there will appear in the tree some nodes that were not in the original sequence; e.g., NEG for unary "-". It is also possible that some nodes -- entire subtrees -- will have been pruned away and replaced with the values returned when special conversion functions were applied to them; e.g., a fixed-to-binary conversion. (Our compilers will keep track of the correspondences between the original sequence of nodes and later sequences to facilitate error recovery and error-message print-out.)

The specific descriptions below should bring these ideas into sharp focus for the reader.

## XPL\_LEXICAL\_ANALYSER

The form of a <translator definition> is much like the form of a <procedure definition>. It has a label, XPL\_LEXICAL\_ANALYSER at the beginning of Figure 1, followed by an <identifier> in parentheses. The <identifier> is the name of the procedure or translator from which the current translator is to get its terminal symbols. Next comes the <translator body> followed by an <ending>. The <ending> is the reserved word END followed in this case by XPL\_LEXICAL\_ANALYSER.

The <translator body> is a list of translation rules that define variables or <nonterminal>s that are local to the <translator>. The form of the translation rules is copied from that used in the transduction grammars of Lewis and Stearns (L&S 68). The part of each rule to the left of "=" is similar to a production of a context-free or BNF grammar and the part to the right is the "transduction element"; i.e., the left half gives the syntax and the right half gives the corresponding translation.

There are, however, two important ways in which our "productions" differ from BNF. First, there is a syntactic difference: whereas in BNF the nonterminals are quoted with the symbols < and > and the terminals are left free standing, we have chosen the programming language convention of quoting the terminals with a pair of single-quotes (') and have left the nonterminals unquoted in concert with identifiers. This choice is important because it facilitates the use of special meta-symbols, such as the vertical bar |, and because it makes our addition to XPL compatible with the existing

lexicon. Second, our "transduction elements" and the right parts of our "productions" are allowed to be regular expressions. The meanings of these translation rules are probably best indicated by specific examples.

/\* FIRST, WE MAKE SOME PRELIMINARY DEFINITIONS.\*/

The first translation rule in Figure 1 indicates that a <binary digit>, or rather a `BINARY_DIGIT`, is either the digit 0 or the digit 1 and that the translation for a 0 is 0 and for 1 is 1. The second rule is similar to the first.

The third translation rule indicates that, if an `OCTAL_DIGIT` is a `BINARY_DIGIT`, then the translation for the former is the same as that of the latter.

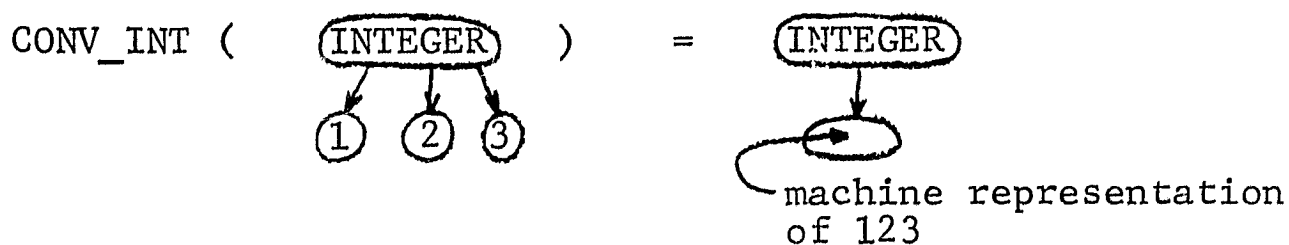
The fourth rule introduces an abbreviation. When the predefined identifier `RIGHT_PART` is used after the "`=>`", the effect is the same as if the right part of the production given to the left of the "`=>`" had been written in the place of `RIGHT_PART`. We did not use this abbreviation in the first three rules for purposes of illustration.

The three dots in the rule defining `LETTER` are used as an expository abbreviation and are not part of our language. Actually, the definitions of `DIGIT` and `LETTER` are unnecessary since these nonterminals will have predefined meanings (`LETTER` will probably include only A through Z). Other identifiers will also be predefined thus giving the user quick access to, for example, the EBCDIC and ASCII character sets and perhaps standard sets of `RESERVED_WORDS` and `OPERATORS`.

The next rule defines SYM\_LESS\_QUOTE to be any EBCDIC character (symbol) except the single-quote. Four adjacent single-quotes are necessary to represent the single terminal symbol, single-quote, due to our use of the PL/I CHARACTER convention to represent terminal symbols.

```
/* NEXT, WE DEFINE THE BASIC "TEXTUAL ELEMENTS" OR "WORDS"
   OF XPL. */
```

The next rule defines an INTEGER to be a string of one or more, by virtue of the +, DIGITs. The translation associated with INTEGER is the result obtained when the function CONV\_INT (convert integer) is applied to a node named INTEGER having as descendants the translations of one or more DIGITs. It is assumed that the value returned by CONV\_INT is a node named INTEGER that has one descendant, a terminal node whose name is the machine representation of the corresponding integer; e.g.,



Several points are to be noted with regard to this translation rule:

First, ubiquitous conversion functions such as CONV\_INT will be predefined in TWS/1.

Second, the % preceding CONV\_INT indicates that the identifier is to be interpreted not as a nonterminal in the current grammar but as an identifier defined globally with respect to the current translator. In general, identifiers

marked by % may be predefined or defined by the language designer, and they may be simple variables of any type, arrays, trees, labels, or procedures.

Third, whereas in the transduction grammars of (L&S 68) the "transduction elements" are meant to indicate translations to strings, our translation parts are meant to indicate translations to trees. Each translation part is interpreted as the Polish suffix representation of the node that is the translation of the corresponding syntactic category. The last symbol in the sequence in the translation part is taken to be the name of the node, and the preceding symbols are taken to specify the descendants of the node. By putting parentheses around a subsequence in the translation part, one can indicate a subtree as one descendant of the main node. As in (L&S 68) there is a one-to-one correspondence between instances of a given symbol in the right part of a production and instances of the same symbol in the corresponding translation part. Normally, when multiple instances of a given symbol appear in both the right part of the production and the translation part, the instances in the translation part are paired with those in the right part in order from left to right; other orders of matching may be indicated via subscripts, as is illustrated below.

A BIT\_STRING is a doublequote followed by either a HEX\_INTEGER or a BIT\_GROUP followed by zero or more, by virtue of the \*, BIT\_GROUPS followed by a doublequote. The associated translation is computed by the function COMBINE, which converts the

various binary, quartal, octal, and hexadecimal digit strings into bit strings and combines them to form one bit string. As in the INTEGER case, we assume that the result returned by COMBINE is a node named BIT\_STRING having one descendant, a terminal node whose name is the machine representation of the final bit string.

The rule defining NUMBER and its translation illustrates another new notion. The "output asterisk" following "=)" indicates that the translation of NUMBER, a node named NUMBER having one descendant, is one of the nodes in the output sequence that is the translation of the input sequence as defined by the current grammar. The order of the nodes in the output sequence is the same as the left-to-right order of the corresponding nodes in the full parse tree associated with the input string. Thus, since our translators are based on left-to-right parsers, the translators will generate the desired translation sequence if they output each translation-node that is marked by an asterisk immediately after the node is constructed; hence the term "output asterisk".

The function NAME\_LIST, used in the definition of IDENTIFIERS and their translations, is assumed to be a predefined function that converts identifiers into their machine representation (via a hash table or the like). It may also provide some compiler-writing conveniences related to postsyntactic computations involving "attributes" of identifiers.

The definition of OPERATORS given here is somewhat different from existing XPL. We have delayed the recognition of the multiple-symbol operators, \*\* and ||, until the XPL\_SYNTACTICAL\_ANALYSER stage. This causes the syntactical grammar of XPL to become LR(2) rather than LR(1); i.e., one state in the corresponding parser must look ahead two symbols rather than just one. This extra look-ahead is easy to implement using the LR(k) techniques of (DeR 69) and it effects only the one state. Alternate translation rules for OPERATOR and OP\_STRING are given in the comment following the XPL\_LEXICAL\_ANALYSER. They define XPL as it is now. It should be clear from these why we chose the former definitions.

```
/* NOW WE DEFINE IGNORABLE SEPARATORS (I.E., "SPACES"). */  
/* FINALLY, WE DEFINE AN XPL TEXT. */
```

Finally, note that all but one of the last ten rules have null translation parts. This is because they are concerned with defining SPACES, including COMMENTS, and indicating where SPACES are allowed and where they are required. The exceptional rule has a translation part which indicates that "\$" inside a comment has special significance; it causes a compiler toggle to be switched.

## XPL\_PREPROCESSOR

The next translator, labelled XPL\_PREPROCESSOR<sup>+</sup>, gets its input from the XPL\_LEXICAL\_ANALYSER and it sends its output to the XPL\_SYNTACTICAL\_ANALYSER. It is this translator that processes the "compile-time facilities" (parameterless macros) of XPL. This processor translates its input string into a rather flat tree that reflects only the grossest block structure of the program; then the procedure SUBST traverses the tree in a left-to-right way making appropriate substitutions as it goes and outputting the resulting sequence to the XPL\_SYNTACTICAL\_ANALYSER.

Since the text to be substituted for a literally declared identifier is represented by a STRING constant, the XPL\_LEXICAL\_ANALYSER must be used by SUBST to translate the text. Provisions will be made for such "abnormal" uses of these translators.

---

<sup>+</sup>Actually the current XPL compiler has no preprocessor; the macros are processed by the XPL syntax analyser. We have separated the macro processing out to illustrate how a language such as PL/I with more complex compile-time facilities might be defined. Also, we have made a slight simplification in the grammar for expository purposes: we have disallowed literal and attribute declarations being given in a single declaration statement.

Finally, we note that it would have been possible for the language designer to increase the efficiency of his resulting compiler by taking advantage of the fact that parsing is done in a left-to-right, bottom-up fashion. He could have associated appropriate procedures (sub-parts of SUBST) with several of the productions defining the preprocessor, so that the substitution process would have happened simultaneously with the parsing, thus obviating the need to build a tree. This would also allow the XPL\_LEXICAL\_ANALYSER, PREPROCESSOR, and SYNTACTICAL\_ANALYSER to execute in parallel. A version of XPL\_PREPROCESSOR that employs this more efficient technique is illustrated in the appendix. The ability to choose between these two extremes will, we believe, prove invaluable in the development of languages and practical compilers; the user will be able to start with a quick and dirty version and gradually tune it to the desired level.

## XPL\_SYNTACTICAL\_ANALYSER

We now proceed to the XPL\_SYNTACTICAL\_ANALYSER. Note that here, as in the above case, there is only one output asterisk in the entire translator; i.e., we have taken the simple approach again: this translator constructs an abstract syntax tree and outputs it only after the entire tree is constructed and all of the syntax analysis is complete.

Again we could have associated appropriate code-generating and bookkeeping routines with each production so that interpretation of the source code or translation to intermediate or object code would have been effected simultaneously with the parsing. If we had one so, the nodes specified in the translation parts of our rules would have served only as argument lists for those routines.

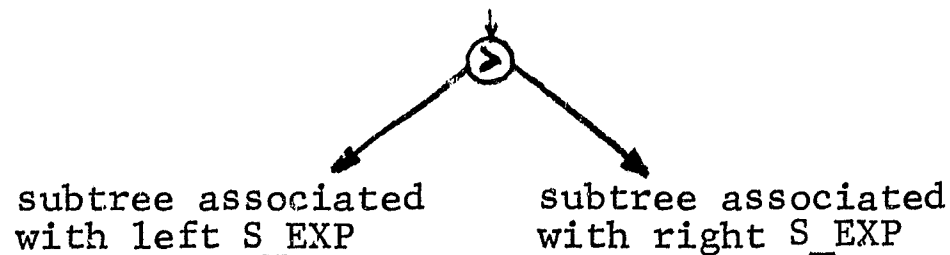
The second translation rule in this analyser emphasizes that, beside the advantage of conciseness, allowing regular expressions as right parts of productions and as translation parts provides the user with an ability to indicate translations which are nodes with arbitrarily many descendants.

Note that the function BUBBLE\_LABELS (not to be pre-defined) is applied to the subtrees indicated in the translation parts of several rules. The intent is that this function "bubble up" through the tree the appropriate information about declaring labels. To answer the question "how high is this information bubbled," and therefore,

"what is the scope of a label," one must investigate the definition of BUBBLE\_LABELS (not given here) and the grammar (to determine which nodes the function is applied to).

Note that in the first alternative in the translation rule for IF\_ST an extra descendant, a terminal node named NO\_OP, is added to the translation-node (for homogeneity) although it has no counterpart in the right part of the production. In general, extra descendants may be added to nodes at will. Such descendants may be used, for example, to associate semantic information with certain nodes; however, we propose below a more palatable scheme for such "semantics" or "tree attributes".

The translation part of the rule defining L\_PRI (logical primary) illustrates another point. When the last symbol in a sequence in a translation part is a nonterminal symbol, that nonterminal must have as its associated translation a terminal node, as is the case with RLN (relation). To construct an appropriate translation, the terminal node is converted to nonterminal by adding to it the indicated descendant. In the case at hand, if RLN is associated with  $\odot$ , then the corresponding translation-node is:



# FEATURES NOT ILLUSTRATED ABOVE

In cases such as the latter where there is more than one instance of a given symbol in the right part of a production, the instances of that symbol in the translation part are usually matched from left to right with the instances in the right part. To indicate a different ordering in the translation part, we subscript the symbols, as in (L&S 68). For example, if we had wanted to reverse the order of the two S\_EXPs above (clearly a silly supposition in this case), then we would have written

$$\begin{aligned} L\_PRI &= S\_EXP \mid S\_EXP \text{ RLN } S\_EXP \\ &\Rightarrow S\_EXP \mid S\_EXP(2) \ S\_EXP(1) \text{ RLN} \end{aligned}$$

A practical example where this feature would be useful can be found in the PAL programming language (Eva 69) in which one can write either

'TEST' L\_EXP 'IFSO' BA\_ST 'IFNOT' BA\_ST  
or 'TEST' L\_EXP 'IFNOT' BA\_ST 'IFSO' BA\_ST

The corresponding translation parts might be

L\_EXP BA\_ST BA\_ST 'IF'  
and L\_EXP BA\_ST(2) BA\_ST(1) 'IF'.

There are two more features not illustrated above that we intend to implement for the sake of generality.

First, the syntax given below allows functions to be applied to the right parts of productions (or parts thereof)

as well as to translation parts. Such functions could be used to effect context-sensitivity and to incorporate semantic conditions into the syntax.

Second, although we have not as yet made provisions in the syntax given below, we would like to allow several translation parts to be associated with each production. This would provide, for example, a convenient way of associating semantic information with nodes of abstract syntax trees and it might provide an easier way of specifying the bubbling of labels up trees. To implement this notion in full generality, however, would be to implement the "tree attributes" of Knuth (Knu 66); it is not clear that this is desirable or useful without the general functional capabilities of LISP. It would be trivial to implement "derived attributes" (information flows up the tree only), but "inherited attributes" may need to be limited in some way. Part of the development still necessary for our language is concerned with making decisions about what are good compromises with regard to the implementation of tree attributes.

#### IV. The Proposed TWS Language: Formal Description (partial)

In Figure 3 we have given, in the language illustrated above, a formal description of the syntactic features which must be added to XPL to extend it to the desired TWS language. Thus, Figures 1 and 3 combined provide a formal description of the lexicon, macros, syntax, and translation to abstract syntax, for our proposed new language, TWS/I. We believe that this concise self-description and the related bootstrapping capabilities of this language are testament to its viability and desirability. We support its feasibility in the next section.

We make only one comment regarding these new translation rules. The function SIMILAR, applied in the translation part of the production defining SYN\_TRANS\_PAIR (syntax, translation pair), is intended to check the right part of each production and its corresponding translation part to see that they have similar structure. This entails determining the correspondences between symbols in the two parts of the rules and outputting error messages when it appears the user may be indicating the discarding of useful information by not using some translations.

#### V. Recent Results that make the TWS Feasible.

Aside from the easily formalizable notion of cascaded grammars presented above, the results reported in the author's dissertation (DeR 69, see also



DeR 70a, b) and those to be reported in a forthcoming paper (DeR 70c)<sup>+</sup> form much of the basis that makes the proposed system feasible.

The dissertation, entitled "Practical Translators for LR(k) Languages," presents techniques for constructing practical parsers, comparable in size and speed to precedence parsers, for the LR(k) grammars of Knuth (Knu 65). It also discusses the conversion of those parsers to string-to-tree translators, as specified by rules much like those illustrated above, and it discusses the use of these techniques in building compilers.

In (DeR 70c), entitled "Extended LR(k) Grammars," the techniques presented in the former will be extended to include productions with regular expressions as right parts. This extension is a simple one because the original techniques involved starting with a finite-state machine and converting it to a deterministic pushdown automaton to parse a given LR(k) grammar. The direct correspondence between finite-state machines and regular expressions allows one to incorporate a finite number of states into the deterministic pushdown automaton to recognize the right parts of productions

---

<sup>+</sup>To be finished in September, 1970. We have developed the techniques and must merely write them up.

and also a similar finite set of states for popping a right part, once recognized, off the top of the pushdown stack.

Also discussed in the latter paper will be the inclusion of the subtraction operation in right parts of productions. This extension is also easy to implement when using LR(k) techniques. First, we construct an LR(k) parser for the grammar as if each "-" were a "|", but we keep track of all states and transitions due partially or wholly to the alternatives that are to be subtracted out and the defines thereof. This first step causes the intersection of the various alternatives to be explicitly represented in the machine. Then we merely remove from the machine the states and transitions that are due solely to the alternatives to be subtracted and those that are due solely to the appropriate intersections.

Preliminary implementations of the simplest results reported in (DeR 69), i.e. parser-generators for the "Simple LR(1)" grammars, at both the University of California at Santa Cruz (DMS 70) and the University of Toronto<sup>+</sup>, seem to substantiate the hand-derived demonstrations in (DeR 69) that these LR(k) techniques are indeed practical. It appears that the techniques are superior to the extended precedence techniques (McK 66) currently used in the XPL system as

---

<sup>+</sup>Personal communication from Professor J. Horning regarding the work of graduate students at Toronto.

regards the time required to construct a parser for a given grammar, and it appears that they are at least comparable as regards the size and speed of the resulting parsers. Of course, with respect to the class of grammars accepted, the extended LR(k) techniques are far superior to any previously existing, highly-efficient, bottom-up techniques, as well as to many inefficient top-down and modified top-down techniques. Further, the LR(k) techniques have the pedagogical advantage of being based on the rigorous areas of automata theory and formal grammars.

A final idea, which we will incorporate into our system, is related to the notion of built-in definitions; e.g. the ones mentioned above for DIGIT and LETTER. We could also supply predefinitions of INTEGER and IDENTIFIER, perhaps even for L\_EXP (logical expression). Two reasonable ways of implementing these predefinitions come to mind immediately. First, we could merely add the appropriate rules to the user's grammar and proceed as usual. Second, we could implement "subparsers" or "subtranslators" for these "subgrammars" once and then have the translator for the user's overall grammar call the appropriate "subtranslator" when it is needed. Since such calling presupposes that the overall translator can determine when it is appropriate to call a given "subtranslator", i.e. when it is time to parse an L\_EXP, say, the notion of deterministic top-down parsing creeps in. We are

led, then, to the ideas of partitioning grammars<sup>+</sup> and of grammars being LL(k) (L&S 68) with respect to the syntactic category that is the root node of a given subgrammar. Our techniques for computing look-ahead sets in the LR(k) case are readily useable in the LL(k) case also; thus, the addition of built-in "subtranslators" for ubiquitous syntactic categories will be easy in our system. Of course, if the user's overall grammar is not LL(k) with respect to a given subgrammar, i.e. if our preexisting subtranslator cannot be used, we can resort to the first implementation alternative mentioned above.

We have already noted that our translation rules, were adapted from the transduction grammars of (L&S 68) and that we intend to extend these rules to include at least some of the generality and convenience of Knuth's "tree attributes" (Knu 66). Thus, not only do our grammars and parsers have a rigorous base, but so also do our translation rules and translators. Further, we believe that our method of language specification is natural and convenient enough that any programmer having minimal exposure to the notions of regular expressions and context-free grammars, and getting a

---

<sup>+</sup>Note the similarity of these comments with the approach of Korenjak (Kor 69) regarding the implementation of practical LR(k) parsers.

quick introduction to transduction grammars and tree attributes, will find it easy to learn to program in the language TWS/I.

## VI. Comparison With Previous TWS Efforts, Evaluation

The reader who is knowledgeable in the TWS area will have noticed some similarities between our system and some previous efforts.

Of course our facilities for postsyntactic language specification are nearly identical to those of existing XPL and primarily amount to procedures written in a PL/I-like language with convenient compile-time facilities, bit-string manipulation facilities, a facility for directly executing machine-language instructions, and a data type for representing and manipulating trees and control structures. We have discussed these features almost not at all, in part because they are not novel and in part because XPL is so well documented.

An important difference between our TWS and META (Sch 64), TMG (McCl 65), COGENT (Rey 65), TGS (Che 65), and CC (BMR 67) is that ours incorporates highly-efficient bottom-up parsing techniques whereas they use relatively slow top-down and modified top-down techniques. In addition to efficiency considerations, top-down techniques usually bar left-recursive productions; this is a severe restriction which does not plague our system.

Regular expressions, or similar notations are employed in META and AED (Ros 67). However, we know of no system that is designed around a notation that is as clean, unified, powerful, rigorously based, and pedagogically sound as our translation rules. Furthermore, with the possible exception of COGENT, our

method of language specification is unique for its separation of syntax and translation parts. For example, TMG and META use statements in which the syntax and translation are thoroughly mixed,<sup>+</sup> resulting in a language specification which is useless as a syntactical reference for the programmer. On the other hand, the syntax parts of our translation rules together make an excellent syntactical reference for the language being defined; one of our TWS/1 compiler options will be a facility for listing these syntax parts separately as a syntactical reference.

Whereas META-II was limited by a lack of facilities in its compilers for reordering their output relative to their input, our system will allow arbitrary reordering through manipulation of trees, for example.

Whereas COGENT allowed the building of data structures based on the syntax only, our system will allow the construction of arbitrary auxiliary structures. Also, our language for describing code generating routines and the like, i.e. XPL, is, we believe, superior to and certainly better documented than COGENT's generator language. Especially with the addition of tree attributes, we believe our TWS will be found useful, as has COGENT been, in the areas of symbolic mathematics and natural language translation and interpretation. For example, the notion of tree attributes is similar to the idea of associating "features" with nonterminals in the systemic grammars of Halliday (Hal 61).

---

<sup>+</sup>Also note that the functions INSTALL and MARKS of TMG are implicit in our system.

Since FSL (Fel 64) was designed as a compiler writer's tool and our system is aimed at language designers, the two systems seem quite different superficially; and yet, there are significant similarities between the two. For example, both systems are based on bottom-up parsing techniques, although our LR(k) techniques are more general and efficient than the Floyd-Evans production-language techniques used in FSL. Also, our viewpoint is well described by Feldman's Figure 12 in (F&G 68), except that we envision several components of language specification in general. For instance, we found it convenient above to have a lexical component, a macro component, a syntax component, etc., for XPL. Our approach would be somewhat more akin to the six components built into the TGS system, the difference again being that we opt for flexibility by allowing the language designer to choose the number of components that is appropriate for his language.

The most important difference between the FSL philosophy and ours, is that, whereas FSL provides special code-generating facilities, we hope ultimately

to obviate the need for these by making them implicit in the implementations of semantic specifications<sup>+</sup> of languages in a manner similar to what we have done for syntactic specifications.

---

<sup>+</sup>There is some reason to hope that we are on the verge of a breakthrough in this area. See for example (Pra 69) and (D&T 70).

One additional point regarding our system is that debugging will be particularly easy (a) because the data type used to represent the inputs and outputs of translators is directly available to the user and (b) because translator definitions can be compiled and tested separately just as procedure definitions can be.

## VII. Summary.

In summary, we have tried to retain the good ideas that have appeared in previous TWS efforts while incorporating the efficient and general parsing techniques that have recently become available. In addition, we have provided in our language a clear syntax based on transduction grammars, and we have designed our system around the unifying idea of cascaded grammars and cascaded translators. We believe that the result is a clear, pedagogically sound language that represents a significant step forward in the TWS area.

```

XPL_PREPROCESSOR:
TRANSLATOR ( XPL_LEXICAL_ANALYSER ):

  PROG = STRING ;
  STRING = GEN_SYMRQL+ ;
  GEN_SYMBOL = GROUP | BLOCK | LITERAL_DEC | BASIC_SYMBOL ;
  GROUP = DC_SEMI STRING 'END'
  DC_SEMI = DO ':'
  DC = 'DO'
  BLOCK = HEAD STRING 'END'
  HEAD = 'BEGIN' | 'PROCEDURE'
          | DC NON_SEMICOLON
  NON_SEMICOLON = 'WHILE' | 'CASE' | 'IDENTIFIER'
  LITERAL_DEC = 'DECLARE' LITERAL ( ',' LITERAL )* ':'
  LITERAL = 'IDENTIFIER' 'LITERALLY' 'STRING'
  BASIC_SYMBOL = 'NUMBER' | 'STRING' | 'IF' | 'THEN'
                | 'BY' | 'IDENTIFIER' | '.' | '*'
                | '+' | '-' | '...' | ':' | ';' | '||'
                | ( 'BEGIN' | 'PROCEDURE' | 'DO' | 'END' | 'LITERALLY' )
                => '*' RIGHT_PART ;

/* ALL NODE NAMES OUTPUT BY THE XPL_LEXICAL_
/* ANALYSER EXCEPT THOSE SPECIFICALLY
/* SUBTRACTED OUT.
*/

END XPL_PREPROCESSOR:

```

```

/*-----*/
/* FIGURE A1. AN ALTERNATE DEFINITION OF THE XPL_PREPROCESSOR THAT RESULTS IN A MORE EFFICIENT COMPILER THAN THE VERSION
/* GIVEN IN FIGURE 1. HERE WE TAKE ADVANTAGE OF THE FACT THAT THE UNDERLYING PARSER IS LEFT-TO-RIGHT, BOTTOM-UP. NOTE THAT
/* THIS GRAMMAR IS SOMEWHAT LESS USEFUL THAN THE PREVIOUS GRAMMAR AS A SYNTACTICAL REFERENCE FOR XPL.
/*-----*/

```

Acknowledgments

Thanks are due Dr. John C. Reynolds of the Argonne National Laboratory for some initial suggestions that led to our exploitation of the idea of cascaded grammars. Thanks also go to Professor McKeeman for his encouragement and for his interest in contributing to the proposed language.

REFERENCES

- Bee 70 Beech, D. A Structural View of PL/I. Computing Surveys 2, 1. March 1970. pp. 33-64.
- BMR 67 Brooker, R. A., Morris, D., and Rohl, J. S. Compiler-compiler facilities in Atlas Autocode. Comput. J. 9 (1967), pp. 350-352.
- Che 65 Cheatham, T. E., Jr. The TGS-II translator-generator system. Proc. IFIP Congress, New York, 1965, pp. 592-593.
- DeR 69 DeRemer, F. L. Practical Translators for LR(k) Languages. Ph.D. thesis, Dept. of E. E., Mass. Inst. of Tech., Cambridge, Mass., Sept. 1970.
- DeR 70a \_\_\_\_\_ . Simple LR(k) Grammars. Submitted to Comm. ACM, June 1970.
- DeR 70b \_\_\_\_\_ . Minimal LR(k) Parsers. In preparation. University of California at Santa Cruz, July 1970.
- DMS 70 \_\_\_\_\_ , McKeeman, W.M., and Sue, J. Simple LR(k) Grammars: Definition and Implementation. Internal report, University of California at Santa Cruz, July 1970.
- D&T 70 \_\_\_\_\_ , and Thomas, R. A Formal Semantic Basis for Programming Languages. (An extention of the  $\lambda$ -calculus that directly models recursion, memory, assignment, labels, and jumps.) In preparation. Univ. of Calif., at Santa Cruz and the Mass. Inst. of Tech., about Dec. 1970.
- Eva 69 Evans, A., Jr. PAL - Pedagogic Algorithmic Language--A Reference Manual and Primer. Dept. of E.E., Mass. Inst. of Tech., Cambridge, Mass., June 1969.
- Fel 64 Feldman, J. A. A formal semantics for computer oriented languages. Ph.D. thesis. Carnegie Inst. of Tech., Pittsburgh, Pa., 1964.
- Hal 61 Halliday, M. A. K. Categories of the Theory of Grammar. WORD 17, 1961.

REFERENCES

- Knu 65 Knuth, D. E. On the translation of languages from left to right. Inf. Contr. 8 (Oct. 1965), pp. 607-639.
- Knu 66                     . Semantics of context-free languages. J. Math. Sys. Th. 2 (February 1966), pp. 127-145.
- Kor 69 Korenjak, A. A Practical Method for Constructing LR(k) Processors. Comm. ACM 12, 11 (Nov. 1969), pp. 613-123.
- Lan 66 Landin, P. J. The next 700 programming languages. Comm. ACM 9 (March 1966), pp. 157-166.
- L&S 68 Lewis, P. M., and Stearns, R. E. Syntax-directed transduction. JACM 15 (July 1968), pp. 465-493.
- McCa 66 McCarthy, J. A formal description of a subset of ALGOL, Formal Language Description Languages (Ed: Steele, T. B.), North Holland Pub. Co., Amsterdam, Holland, 1966.
- McCl 65 McClure, R. M. TMG--a syntax-directed compiler. Proc. ACM 20th Natl. Conf., 1965, pp. 262-274.
- McK 66 McKeeman, W. M. An Approach to Computer Language Design. Ph.D. thesis. Stanford University, Stanford, California, 1966.
- MHW 70 McKeeman, W. M., Horning, J. J., and Wortman, D. B. A Compiler Generator. Prentice-Hall, Inc., Englewood Cliffs, N. J., Aug. 1970.
- Pra 69 Pratt, T. A hierarchical graph model of the semantics of programs. Proc. SJCC 34 (May) 1969, pp. 813-825.
- Rey 65 Reynolds, J. C. An introduction to the COGENT programming system. Proc. ACM 20th Natl. Conf., 1965, pp. 422-436.

REFERENCES

- Ros 67            Ross, D. T. The AED approach to generalized  
                 computer-aided design. Proc. ACM 22nd Natl.  
                 Conf. 1967, pp. 367-385.
- Sch 64            Schorre, D. V. META-II: A syntax-oriented  
                 compiler writing language. Proc. ACM  
                 19th Natl. Conf., 1964, p. D1.3.