The XPL compiler generator system*

by W. M. McKEEMAN

University of California
Santa Cruz, California

and

J. J. HORNING, E. C. NELSON
and D. B. WORTMAN

Stanford University
Stanford, California.

Objectives and results

The development of the system described here was originally motivated by the need to develop a good student language compiler for a large IBM System/360. An examination of the tools and methods available caused us to establish the subgoal of developing a translator writing system in which we could prepare the student compiler. In our opinion, then and now, the total effort was smaller, and the end product better, for the combined project than for the original project using previously available tools.

In the belief that this system will be useful to others who are interested in language development, we have extended it beyond our immediate application and are making it generally available. We hope that this will help dispel the myth that the generation of new compilers is necessarily a long and arduous task, consuming many man-years of effort. Students in the systems programming course at Stanford have been using this method to write operating translators in three to six weeks.

We made two major decisions in designing our TWS: the choice of a compilation method and the choice of a language in which to express the translators. We selected a bottom-up parsing algorithm rather than recursive descent since the IBM System/360 is ill-suited to recursion. In any case, code generation is easier with bottom-up methods. The particular method used is an extension of several which have been previously reported [Floyd 63] [Wirth and Weber 66] [McKeeman 66].

Compilers have been written in assembly languages, "general purpose" high-level languages, and special purpose "compiler compiler" languages. We rejected the first as being completely beyond our available manpower to program, debug, or maintain. Our choice was a compromise between the second and third options. The language XPL contains many features which are sufficiently "close to the machine" for efficient translators, and omits features not required for translation, yet retains much of the flexibility and style of PL/I.

Our TWS consists of four major programs: XCOM, XPLSM, ANALYZER, and SKELETON. XCOM is a one-pass compiler from XPL into System/360 machine language. Since it is written in XPL (it consists of about 3500 source cards) it is self-compiling on a System/360.

XPLSM, the interface between XPL programs and OS/360, is a small assembly language program which handles program loading and input/output using OS/360 data management methods.

ANALYZER accepts a BNF* grammar, checks it for compatibility with the parsing algorithm, and prepares parsing decision tables in the form of XPL declaration statements with the initial attribute. The program is written in XPL (1400 cards).

SKELETON is a proto-compiler, which when supplied with tables from ANALYZER becomes a table-driven syntax checker, to which code emitters may be added to create a compiler.

---

*This work was supported by the Stanford University Computer Science Department.

*Backus-Naur Form or Backus Normal Form
The method.

The translation method depends upon the well-known fact that the phrase-structure of a BNF-described source language can be used to direct the production of object code [Wirth and Weber 66]. All of the following phrases apply to this method:

- table-driven,
- syntax-controlled,
- bottom-up,
- one-pass,
- left-to-right,
- no-backup, and
- canonical parse.

The general parsing algorithm, depicted in Figure 1, consists of two cycles corresponding roughly to the actions of:

1. Recognizing and stacking the basic symbols (identifiers, numbers, operators, etc.) of the source language.
2. Substituting a phrase class name for a phrase and emitting the associated object code.

Within the inner loop there is a call on the text scanning procedure. The responsibility of the scanner is to recognize the symbols on the lowest level of detail in the language (identifiers, reserved words, etc). The scanner is a source language dependent routine that is programmed with the text manipulation features of XPL. As a result the most frequent decisions (those on individual characters) are not made by the general parsing algorithm for reasons of efficiency.

The success of the general parsing algorithm depends upon how generally, and how efficiently the decisions indicated in Figure 1 can be made. The stacking decision is made on the symbol pair consisting of the top of the stack (where any symbol is possible) and the input symbol (where only terminal symbols can occur). There are four possibilities:

1. The input symbol is stacked and the question is asked again for the next symbol.
2. The input symbol is not stacked, instead a reduction is made and (perhaps) some code emitted.
3. The symbol pair does not contain enough information upon which to make the decision.
4. A syntactic error in the source text is discovered and error recovery initiated.

When the pair fails to contain enough information (3), an auxiliary table of triples, which correspond to the two top elements of the stack and the input symbol, is consulted. Since the number of decisions which must be made using triples is small for many useful grammars, the necessity for searching a table need not slow down the compiler appreciably. The current XPL grammar has 89 symbols, 42 of which are terminal so that the symbol pair table has 37382-bit entries. Of these 3175 represent illegal combinations (syntax error), 239 indicate that the input symbol should be stacked, 306 indicate that it should not be stacked, and 18 represent conflicts for which the decision must be made using triples, of which there are 165.

Another series of tables, equivalent to (1,1) context [Floyd 64], is used to decide which reduction to make once the reducible string is located on the top of the stack. These tables are automatically produced from the BNF description of the source language, as described in section 4.

It can be shown, based upon these tables, that if the general parsing algorithm proceeds to normal completion (i.e., to the point where all the source text has been
The language XPL

Translation methods are not “all alike,” and we do not suggest that all translators should be written in the same language. XPL was designed for a particular class of translators that require very elementary kinds of data structure and control. Aside from the problem of text manipulation almost any existing procedural language is sufficient and in fact, considering our bias towards simplicity, overly elaborate.

There are no startling innovations in the following description. One reason is that the efficiency of translators is an important concern, so constructs reflect the structure of the IBM System/360. A second reason is that the technique used in the recognition phase of the translator was first presented [Floyd 63] with the explicit purpose of being efficiently implementable on existing conventional machines.

Before getting into the details, we can describe the basic concepts which must underlie any computer language design. We must have:

1. selection by name (the idea of a set);
2. selection by position (the idea of an ordered set);
3. sub-definition (definition of something in terms of its components);
4. iteration;
5. meaningful primitive operations; and
6. control over the allocation of computer resources.

We expect translators particularly to be maintained, debugged, modified, studied and copied by many people besides the original authors. So we insist that the language be conducive to readable statements of translators.

We want the user to be able to easily follow the logic of a program at two levels. First, he should be able to quickly determine the major structural units, their intent, and the flow of control between them. Second, it should be convenient for him to determine precisely what operations are being performed on the data at any given point. It should be possible to break the program into units both logically (procedures) and visually (paragraphing).

The programmer should be able to access the memory efficiently by operating on the directly addressable quanta of memory (characters, words, etc.). We also require the ability to describe and operate on the basic units of information (bits) both for the generation of arbitrary object code and for efficient packing and unpacking of various tables.

Four kinds of data are particularly useful for translators: bits for recording yes-no decisions; strings of bits for packing information; integers for arithmetic; and strings of characters to represent text, such as the source input, the output program listing, diagnostic messages, and identifiers.

Many of the operations within a translator require the evaluation of integer arithmetic expressions. Our language must include the arithmetic operations of addition, subtraction, multiplication, division and remainder. We also need the logical operations (&, |, ~, and shifting) on either single bits (logical expressions) or on groups of bits (masking operations). Within each data type we must have comparison operations (<, ≤, ≥, ±, etc.) and we need to convert data from one type to another (for instance, from integers to character strings for output).

Text (strings) presents additional problems. Input text must be tested and separated into constituent strings of varying length (e.g., identifiers). The output listing must generally be built up by joining shorter strings. The basic string operations are those of substring selection and concatenation, as well as the extraction of internal codes for the characters.

We would, of course, expect the writer of a translator to be able to provide for emitting the code for any machine instruction. What is frequently true, is that he may want to cause the execution of any instruction during compilation. The constraints placed upon the translator by the machine itself are already severe; it is unwise to compound them by further restricting the translator to some subset of the machine’s capabilities. No single translator is likely to use the full instruction repertoire of a large scale computer but there is no instruction that we can classify a priori as unusable. The simplest example of this kind of requirement is the floating point operations. The compilation mechanism itself has no need for them, but if the language being compiled has floating point values, it is convenient to use the floating point hardware to convert input numbers. Rather than add a type to the compiler language, we make a general provision for instruction-by-instruction, in-line execution of arbitrary machine code.

A basic concept is selection from an ordered set. For

*Even in translation of languages involving floating point numbers, floating point expressions are not frequently used.
ease of expression and efficiency of operation both data and instructions are usually arranged so that the appropriate element can conveniently be selected from a set of alternatives. Arrays with one subscript position are the simplest form representing this construct. Out of arrays we can build stacks (for the syntactic recognition algorithm) and tables (for symbols, types, forward references and the like) and like spaces (for lists, sorting, etc). We do not find sufficient use for more complex constructs (multiple subscripts, queues, explicit lists, PL/I structures) in our translators to justify their implementation. Similarly, it is possible to select among alternative instruction sequences dependent on the value of an expression. In the most common case the expression will be logical (two-valued) in nature and the set of instruction sequences will contain two elements. The usual representation for this action is the IF-THEN-ELSE conditional statement and the CASE statement. [Wirth and Hoare 66]

Repetition of selected sequences of operations is another basic requirement. Two forms of repetition seem particularly important. We may wish to repeat the operations a specific number of times (often with specific values for a controlled variable) or we may wish to repeat them as long as a particular condition is satisfied.

Transfer of control also takes two forms. We may temporarily transfer to a subprogram (possibly supplying it with some parameters) which will return to the original instruction sequence (procedures and functions). Occasionally, however, it is desirable simply to enter the instruction sequence at some other point, with no provision for return (GO TO together with labels).

We chose PL/I as a base language for several reasons. It contains most of the features we require. It is widely known and will probably be the next dominant programming language. The growth of the PL family of languages will require the development and refinement of PL.

There are several distinct reasons for not choosing PL/I itself for the description of translators. First, it does not contain quite all the features we require. More important, however, is that it is a very large language and the machine we are using (IBM System/360) is ill-suited to its full implementation. Consequently all existing compilers for the full PL/I are very large complex programs which produce code unacceptably inefficient for this application. For the same reasons, we do not recommend attempting a compiler for full PL/I written in XPL.

We present here several program fragments written in XPL to give the reader a better feel for the language.

1) A simple indirect sort:

```plaintext
/* SORT IN ASCENDING ORDER */
K, L = ND;
DO WHILE K <= L;
    L = -1;
    DO I = 1 TO K;
        IF (DESCRIPTOR(DX(I)) & MASK) > (DESCRIPTOR(DX(I)) & MASK) THEN
            J = DX(I); DX(L) = DX(I); DX(I) = J;
            L = L - 1;
        END;
    END;
END;
END;
```

2) Code emission procedure from a translator:

```plaintext
PROCEDURE (L,N);
/* L IS THE LEXIC LEVEL, N IS THE ORDER NUMBER */
DECLARE L FIXED, N FIXED;
IF (L < 3 I L = LL) AND N < 16 THEN
    DO; /* A NAME */
        IF L = LL THEN L = 3;
        CALL EMIT(SIN(L,N))
    END;
ELSE
    CALL EMIT_NAME(L,L,N); /* A LITERAL */
END EMIT_NAME ;
PROCEDURE (L,N);
/* L IS THE LEXIC LEVEL, N IS THE ORDER NUMBER */
DECLARE L FIXED, N FIXED;
IF (L < 3 I L = LL) AND N < 16 THEN
    DO; /* A NAME */
        IF L = LL THEN L = 3;
        CALL EMIT_3(NAAE_OP,L,N)
    END;
END;
```

3) Instruction interpretation routine from an interpreter:

```plaintext
PROCEDURE (SP);
/* CASE 1 SHOW EQUAL OPERATOR */
DO;
    SP = SP+24; /* PUSH THE SPACE */
    IF (DESCRIPTOR(SP) AND ON VALUES) THEN
        DO CASE SSH(SP,4) ;
            RUN_VALUE(SP) = DISPLAY + {OP & "OF"} ;
            RUN_VALUE(SP) = DISPLAY + {OP & "OF"} ;
            RUN_VALUE(SP) = DISPLAY + {OP & "OF"} ;
            END;
        END Type(SP) = INDIRECT_TYPE ;
    END;
END SP;
```

4) Printing the date:

```plaintext
DECLARE DATE_LITERAL LITERALLY 'OUTPUT(1) = 1';
DECLARE DATE_LITERAL LITERALLY 'OUTPUT(1) = 0';
PROCEDURE (MESSAGE, D);
DECLARE MESSAGE CHARACTER, D FIXED;
DECLARE MONTH(12) CHARACTER INITIAL ('JANUARY', 'FEBRUARY', 'MARCH', 'APRIL', 'MAY', 'JUNE', 'JULY', 'AUGUST', 'SEPTEMBER', 'OCTOBER', 'NOVEMBER', 'DECEMBER'),
DAYS(365) FIXED INITIAL (0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335);
DECLARE YEAR FIXED, DAY,FIXED;
YEAR = D/1000 + 1900;
DAY = D MOD 1000;
IF (DAY & 3) = 0 THEN IF DAY > 59 THEN WN = WN + 1;
N = 11;
DO WHILE DAY < DAYS(WN); N = N + 1; END;
OUTPUT = MESSAGE || MONTH(WN) || ' ' || DAY-DAYS(WN) || ' ' ||
YEAR || ' ';```
XPL is formally defined by the BNF grammar given in the Appendix.

Upon superficial examination, XPL does not appear to differ significantly from PL/I. A program consists of a sequence of statements; the effect of a program is determined by executing those statements in order. (An XPL program is equivalent to the body of a PL/I external procedure with option MAIN). Two types of statements—declarations and procedure definitions—cause no action when executed but rather affect the meaning of other statements in the program. The remainder of the statements are imperative in nature, causing the computation and moving of values (assignments), repetitive and selective execution of statements (groups, if statements), invoking subprocesses (call statement and function designators), terminating subprocesses (return), and absolute transfer of control (go to).

A brief list of the differences between XPL and the PL/I constructs with the same form follows:

1) All variables must appear in a declaration before they appear in any other statement (thus declaration is mandatory).
2) The lower bound of all arrays is implicitly zero.
3) Abbreviations are not pre-defined (use CHARACTER not CHAR).
4) Only data types FIXED, CHARACTER, and BIT are provided.
5) Attributes cannot be factored.
6) Bit strings are substantially different.
7) Character strings all have the attribute VARYING, a maximum length of 256, and start with character zero (not one). Concatenation is relatively expensive (slow), BYTE, SUBSTR, and LENGTH are fast. SUBSTR may not appear on the left of an assignment statement.
8) DO loops have only positive steps.
9) Procedures are nonrecursive and have only value (evaluated) parameters.
10) Structural words such as DO and IF are reserved, and may not be used as identifiers.

Extensions to PL/I that appear in XPL

A simple kind of parameterless macro is provided by the LITERALLY attribute. When an identifier is so declared, then any later occurrence of that identifier will be literally replaced during compilation by the character string following the reserved word LITERALLY. Among its uses are the naming of constants (both for mnemonic reasons and ease of maintenance where many copies of a constant might have to be changed), the introduction of abbreviations for reserved words, and redefinition of identifiers where multiple names are desired to have a single meaning. The following programs are equivalent.

```
DECLARE A LITERALLY '20';
DECLARE B(A) FIXED,
    C(A) FIXED;
B(A) = C(A) + A;
```

```
DECLARE B(20) FIXED,
    C(20) FIXED;
B(20) = C(20) + 20;
```

An alternative notation for 32 bit constants is provided by bit strings with binary, quartal, octal or hexadecimal significance as in: "AB9 (3)57 (2)32 (1)111." Bit strings are delimited by the double quote (" ). The parenthesesized digit at the head of each bit group is interpreted as a field-width-per-digit of that group. Thus for any bit group, the total number of bits represented is (number of digits) * (field width). If there is more than one bit group, the rightmost group is right justified in the 32 bit representation, the next group is right justified in the remaining space, and so on. Note that the first entry in a bit list may be a hexadecimal integer. This construct represents a default field width of hexadecimal significance. All of the following have the same internal machine form: "FF," "(4)FF," "(1)1111 1111," "(3)7 (2)33 (1)1," and 255: that is the 32 bit value 00000000000000000000000011111111. The operators &, |, and ~ can be used to perform masking operations on 32 bit quantities for data packing and unpacking. Two built-in functions SHL and SHR provide access to the hardware logical shift instructions.

The built in function BYTE(S,I) where S is a string and I an integer returns the internal representation of the Ith character in the string, thus providing an explicit conversion from EBCDIC characters to small integers.

Several pseudo variables are available to make input and output as simple and painless as possible. The pseudo variable INPUT has as its value a string representing the next input record. Assigning a value to the pseudo variable OUTPUT as in the CASE statement example below causes a string (conversion from type FIXED or BIT to type CHARACTER will be done if necessary) to be output as the next record on an associated output device. The pseudo array FILE is used to
access scratch storage on direct access devices. FILE (I,J) specifies the Jth record on the Ith file. FILE may appear on the left or right side of an assignment statement causing a device-dependent number of bytes to be written or read from the selected record and file.

For example:

DECLARE X (900) FIXED;
/* ASSUME 2311 DISKS*/
X = FILE (1,K);
FILE (3,K + 1) = X;

transfers a record from one file to another.

Generalizing the selective capability of the IF statement, the CASE statement allows the selection of any one as sequence of statements. The expression following CASE is evaluated and used as an index to select one of the statements in the group body counting from zero, starting from the top. For example:

I = 2;
DO CASE I;
    OUTPUT = 0;
    OUTPUT = "1";
    OUTPUT = 2;
    OUTPUT = '3';
END;
OUTPUT = 4;

will cause 2 and 4 to be printed.

The ability to execute arbitrary machine instructions is provided by the pseudo function INLINE. The arguments of this function are placed directly into the code stream at the point of the function call. An option is provided whereby XCOM will calculate the proper base and displacement fields from the names of the variables. An example of the use of INLINE to do floating point arithmetic in an interpreter written in XPL is given below.

DO; /* FLOAT ARITHMETIC */
/* FIRST LOAD THE FLOATING REGISTER WITH BY */
CALL INLINE("10", 0, 0, BV); /* LE 0, BV */
DO CASE OP - ADD OP; /* NOW EXECUTE A FLOATING INSTRUCTION */
    CALL INLINE("11", 0,0,AV); /* ADD 0, AV */
    CALL INLINE("12",0,0,AV); /* SE 0, AV */
    CALL INLINE("13",0,0,AV); /* ME 0, AV */
    CALL INLINE("14",0,0,AV); /* DE 0, AV */
END;
/* NOW STORE THE RESULT IN BY */
CALL INLINE("15",0,0,BV); /* ST 0, BV */
END;

One should note which capabilities of the machines are missing from XPL. By their absence we are marking them as irrelevant to our translation process and thereby simplifying both our language and its translator. It is worthwhile to note that the addition of features to a language is not a linear process as far as the translator is concerned. Constructs interact with each other, and the addition of a single feature may, in a bad case, double the size of the translator. The potentially exponential growth of translator size with increasing language complexity has two important implications. First, a translation method which works well for deliberately simple test languages or machines may fail for practical languages or existing machines. Second, an elaborate language (we have in mind the full PL/I) may need a much larger translator than needed by several smaller languages which have in aggregate the same features.

Programming in BNF.

Just as we need a language for the description of translators, we need a metalanguage for the description of languages to be translated. BNF has become the most widely used formal metalanguage for programming languages; it is a concise, readable, and unambiguous way to express any context-free grammar. We have adopted it for use in our system.

Each language which has a context-free grammar has arbitrarily many context-free grammars. The art of programming in BNF is to impose additional criteria to select an optimal grammar for use in a translator. Some general requirements are:

(1) The grammar must be compatible with the parsing algorithm used by the translator, which implies (2).
(2) The grammar must be unambiguous, so that each sentence will have a unique phrase structure.
(3) The structure assigned to each sentence should correspond to the "intended" interpretation of the language.
(4) The grammar should permit easy association of code generation with the canonical parse.

Requirements (1) and (2) concern properties which a given grammar either does or does not possess, while (3) and (4) are (at the present state of the art) still matters of judgment and degree.

One program of our system (ANALYZER) builds decision tables for the parsing algorithm from a BNF grammar. In the process, it determines whether the grammar meets condition (1), and if not, gives error diagnostics to pinpoint the problem. Basic to the ANALYZER algorithm is the concept of "running the parser backwards." Instead of successively reducing a string of text to a goal symbol, it successively produces text from the goal symbol, tabulating all the decisions which would be required to parse the produced texts.
Expansions continue until all possible decisions (for the chosen complexity of tables) have been recorded.

Ideally ANALYZER would prepare correct tables for every grammar which users fed it. However, people don’t write good grammars. For one thing, they usually start out ambiguously (even the Algol 60 Committee created an ambiguous grammar). After the ambiguities have been removed, other changes may be necessary to make the grammar compatible with the specific parsing algorithm.

One criterion in designing our parsing algorithm was that it be compatible with a wide class of grammars. However, speed and space considerations required that the context considered in decisions be limited. This, in

FIGURE 2
GRAMMAR ANALYSIS -- STANFORD UNIVERSITY -- ANALYZER VERSION OF JULY 30, 1968.

TODAY IS JULY 30, 1968.

PRODUCTIONS

1. <PROGRAM> ::= _!_ <BOOLEAN EXPRESSION> _!_
2. <BOOLEAN EXPRESSION> ::= BOOLEAN IDENTIFIER := <BOOLEAN EXPRESSION>
   | <ARITHMETIC EXPRESSION> = <ARITHMETIC EXPRESSION>
3. <ARITHMETIC EXPRESSION> ::= ARITHMETIC IDENTIFIER := <ARITHMETIC EXPRESSION>
   | <ARITHMETIC EXPRESSION> - <ARITHMETIC TERM>
   | <ARITHMETIC TERM>
4. <ARITHMETIC TERM> ::= ARITHMETIC IDENTIFIER
   | ( <ARITHMETIC EXPRESSION> )
   | # ( <BOOLEAN EXPRESSION> )

TERMINAL SYMBOLS NONTERMINALS

1. =
2. -
3. ( 10. <PROGRAM>
4. )
5. #
6. :=
7. _!_
8. BOOLEAN IDENTIFIER
9. ARITHMETIC IDENTIFIER

<PROGRAM> IS THE GOAL SYMBOL.

*** ERROR, GRAMMAR IS AMBIGUOUS.

IT IS LEFT AND RIGHT RECURSIVE IN THE SYMBOL <ARITHMETIC EXPRESSION>
turn, limits the class of acceptable grammars, but the limitations cannot be expressed in terms of simple restrictions on the form of the grammar.

Since debugging a grammar can often be a chore, much of ANALYZER is devoted to the generation of rather complete diagnostic messages for the various error conditions. The computer output in Figure 2 was produced by ANALYZER for a typical small grammar.

After printing the grammar, ANALYZER lists its terminal symbols, nonterminal symbols (phrase class names) and goal symbols. The grammar is checked for being left and right recursive in any symbol (one of the obvious, yet frequent causes of ambiguity) [this check was suggested by Don Knuth]. In this example there are

FIGURE 3
GRAMMAR ANALYSIS -- STANFORD UNIVERSITY -- ANALYZER VERSION OF JULY 30, 1968.

TODAY IS JULY 30, 1968.

PRODUCTIONS

1. <PROGRAM> ::= _1_ <BOOLEAN EXPRESSION> _1_

2. <BOOLEAN EXPRESSION> ::= BOOLEAN IDENTIFIER := <BOOLEAN EXPRESSION> 
    | <ARITHMETIC EXPRESSION> = <ARITHMETIC EXPRESSION>

3. <ARITHMETIC EXPRESSION> ::= ARITHMETIC IDENTIFIER := <ARITHMETIC EXPRESSION> 
    | <ARITHMETIC TERM>

4. <ARITHMETIC TERM> ::= <ARITHMETIC TERM> - <ARITHMETIC PRIMARY> 
    | <ARITHMETIC PRIMARY>

5. <ARITHMETIC PRIMARY> ::= ARITHMETIC IDENTIFIER 
    | ( <ARITHMETIC EXPRESSION> ) 
    | # ( <BOOLEAN EXPRESSION> )

TERMINAL SYMBOLS   NONTERMINALS

1. =
2. -
3. ( 
4. ) 
5. #
6. :=
7. _1_ BOOLEAN IDENTIFIER
8. _1_ ARITHMETIC IDENTIFIER
9. <PROGRAM> IS THE GOAL SYMBOL.
FIGURE 3

Cl MATRIX FOR STACKING DECISIONS:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>-</td>
<td>(</td>
<td>)</td>
<td>#</td>
<td>:=</td>
<td>_</td>
<td>BOOLEAN IDENTIFIER</td>
<td>ARITHMETIC IDENTIFIER</td>
<td>&lt;PROGRAM&gt;</td>
<td>&lt;ARITHMETIC TERM&gt;</td>
<td>&lt;BOOLEAN EXPRESSION&gt;</td>
<td>&lt;ARITHMETIC PRIMARY&gt;</td>
<td>&lt;ARITHMETIC EXPRESSION&gt;</td>
</tr>
</tbody>
</table>

TABLE ENTRIES SUMMARY:

83 22 Y 16 N 5 #

Cl TRIPLES FOR STACKING DECISION:

1 N FOR = <ARITHMETIC EXPRESSION> )
2 Y FOR ( <BOOLEAN EXPRESSION> )
3 Y FOR ( <ARITHMETIC EXPRESSION> =
4 Y FOR ( <ARITHMETIC EXPRESSION> )
5 N FOR := <BOOLEAN EXPRESSION> )
6 N FOR := <BOOLEAN EXPRESSION> |

*** ERROR, STACKING DECISION CANNOT BE MADE WITH (2,1) CONTEXT.
7 # FOR := <ARITHMETIC EXPRESSION> =
8 N FOR := <ARITHMETIC EXPRESSION> )
9 Y FOR _ _ <BOOLEAN EXPRESSION> _ _
10 Y FOR _ _ <ARITHMETIC EXPRESSION> =
11 N FOR <BOOLEAN EXPRESSION> _ _ _ _

50 ENTRIES FOR 11 TRIPLES.

TABLE ENTRIES SUMMARY:

5 Y
5 N
1 #
PRODUCED HEAD SYMBOLS:

1 =
2 -
3 (  
4 )
5#
6 :=
7 1
8 BOOLEAN IDENTIFIER
9 ARITHMETIC IDENTIFIER
10 <PROGRAM>
11 <ARITHMETIC TERM>
12 <BOOLEAN EXPRESSION>
13 <ARITHMETIC PRIMARY>
14 <ARITHMETIC EXPRESSION>

FIGURE 3

1111
12345678901234
+-------------------+
| Y | Y | Y | Y |
| Y | Y | Y | Y |
| Y | Y | Y | Y |
| Y | Y | Y | Y |
+-------------------+

CONTEXT CHECK FOR EQUAL AND EMBEDDED RIGHT PARTS:

THERE ARE 12 AND 12 VALID CONTEXTS, RESPECTIVELY, FOR
6 <ARITHMETIC TERM> ::= <ARITHMETIC TERM> - <ARITHMETIC PRIMARY>
7 <ARITHMETIC TERM> ::= <ARITHMETIC PRIMARY>
THEY CAN BE RESOLVED BY LENGTH.

ANALYSIS OF (2,1) CONFLICTS:

THE TRIPLE := <ARITHMETIC EXPRESSION> = MUST HAVE THE VALUE N FOR
4 <ARITHMETIC EXPRESSION> ::= ARITHMETIC IDENTIFIER := <ARITHMETIC EXPRESSION>
IN THE CONTEXT ( ... =
IN THE CONTEXT := ... =
IN THE CONTEXT _ | _ ... =

THE TRIPLE := <ARITHMETIC EXPRESSION> = MUST HAVE THE VALUE Y FOR
3 <BOOLEAN EXPRESSION> ::= <ARITHMETIC EXPRESSION> = <ARITHMETIC EXPRESSION>
IN THE CONTEXT := ... )
IN THE CONTEXT := ... _ | _

ANALYSIS COMPLETE FOR ITERATION 1
* ONE ERROR WAS DETECTED
GRAMMAR MODIFICATION TO ATTEMPT TO RESOLVE CONFLICTS:

11 <:=1> ::= =
12 <:=2> ::= =:
4 <ARITHMETIC EXPRESSION> ::= ARITHMETIC IDENTIFIER <:=2> <ARITHMETIC EXPRESSION>

PRODUCED HEAD SYMBOLS: PAGE 1 OF 1

CONTEXT CHECK FOR EQUAL AND EMBEDDED RIGHT PARTS:

THERE ARE 3 AND 4 VALID CONTEXTS, RESPECTIVELY, FOR
12 <:=2> ::= =: THEY CAN BE RESOLVED BY (1,0) CONTEXT.
11 <:=1> ::= =: THEY CAN BE RESOLVED BY LENGTH.

THERE ARE 14 AND 14 VALID CONTEXTS, RESPECTIVELY, FOR
6 <ARITHMETIC TERM> ::= <ARITHMETIC TERM> - <ARITHMETIC PRIMARY>
7 <ARITHMETIC TERM> ::= <ARITHMETIC PRIMARY>
THEY CAN BE RESOLVED BY LENGTH.
FIGURE 4

C1 MATRIX FOR STACKING DECISIONS:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>:=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>BOOLEAN IDENTIFIER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>ARITHMETIC IDENTIFIER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>&lt;PROGRAM&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>&lt;ARITHMETIC TERM&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>&lt;BOOLEAN EXPRESSION&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>&lt;ARITHMETIC PRIMARY&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>&lt;ARITHMETIC EXPRESSION&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>::=1&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>::=2&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE ENTRIES SUMMARY:

94 Y
20 N
5 #

C1 TRIPLES FOR STACKING DECISION:

1 N FOR = <ARITHMETIC EXPRESSION>)
2 Y FOR ( <BOOLEAN EXPRESSION> )
3 Y FOR ( <ARITHMETIC EXPRESSION> =
4 Y FOR ( <ARITHMETIC EXPRESSION> )
5 Y FOR = <BOOLEAN EXPRESSION> =
6 Y FOR = <ARITHMETIC EXPRESSION> =
7 N FOR <BOOLEAN EXPRESSION> =
8 N FOR ::=1> <BOOLEAN EXPRESSION>)
9 N FOR ::=1> <BOOLEAN EXPRESSION> =
10 Y FOR ::=1> <ARITHMETIC EXPRESSION> =
11 N FOR ::=2> <ARITHMETIC EXPRESSION> =
12 N FOR ::=2> <ARITHMETIC EXPRESSION>)

55 ENTRIES FOR 12 TRIPLES.

TABLE ENTRIES SUMMAR Y:

6 Y
6 N
0 #
we can remove this ambiguity by inserting another phrase class, and separating the recursions involving := and -. The output of a successful run on the revised grammar is given in Figure 3.

After listing the grammar, the terminal, non-
terminal, and goal symbols, ANALYZER has printed the matrix of produced 1-heads (X is a produced 1-head of A if a string starting with X can be produced from A). After computing all valid contexts of all productions, it has checked equal and embedded right parts to determine the context necessary to distinguish them. (If two productions could not be distinguished using one symbol of context to the left and the right, it would have listed all of the problem contexts.) It then computed and printed the stacking decision matrix and the triples for the resolution of pair conflicts. Where the stacking decision could not be made based on the top two symbols of the stack plus the incoming symbol, it flagged the triple and listed the problem production and contexts.

One of the principal problems for bounded context parsing algorithms is the "insulating comma"—that is, a symbol which occurs in many contexts as punctuation, and does not itself serve as adequate context for various decisions [Lynch 68]. In our example, as a result of the use of := in both arithmetic expressions and boolean expressions, one stacking decision cannot be made with the triples table. Frequently this problem is solved by requiring the scanner to make the decision and return <comma1>, <comma2> ... (or <:=1>, <:=2>) as determined by some more global and ad hoc context. An alternative solution for an algorithm which can handle equal right parts is to leave the comma as a terminal symbol and add to the grammar production of the form

\[
<\text{comma1}> : : =, \\
<\text{comma2}> : : =, \\
\ldots
\]

as required. This places the burden of decision on the general parsing algorithm. The process can be partially automated and ANALYZER contains a procedure which will optionally modify the grammar in this fashion to remove conflicts. In the example above, two new nonterminal symbols, and two new productions have been added to the grammar. Subsequent output in Figure 4 shows that this process has removed the local ambiguity from the grammar.

Finally, if desired, the tables in the form required for the SKELETON program may be listed or punched. The tables for the example are shown in Figure 5.

Building a translator.

Once the BNF grammar for the source language has been debugged, the construction of the translator can begin. When the tables punched by ANALYZER are inserted into SKELETON, (and its scanner modified to reflect the identifier, comment, etc. conventions of the new language), SKELETON becomes the XPL description of a table-driven syntax checker for the source language. The new program can then be compiled by XCOM and run under XPLSM. To turn this prototype into a translator, emitters for the object language must be inserted into the case statement for the various productions (refer to Figures 1 and 6). An example of some simple semantic routines to compile the language of section 4 for a single-address, single accumulator machine is given in Figure 7. The compilation method does not restrict the form of the object code. XCOM produces absolute machine language as output; another compiler we have written (for the student language) produces an intermediate code similar to Polish postfix, which is then interpreted. Other choices are possible.

What we have described is a fully operational system which has been in use at Stanford since 1967. Although we are continuing to experiment and improve, we plan to release a stabilized version soon through the SHARE organization. Preliminary documentation, which will ultimately appear in the form of a book, is available from the authors.

REFERENCES

1 J.A. FELDMAN and D. GRIES  
Translator writing systems  
Comm ACM 11 Feb 1968

2 R W FLOYD  
Operator precedence and syntactic analysis  
J ACM 10 July 1963 pp 316–333

3 ———, Bounded context syntactic analysis  
Comm ACM 7 Feb 1964 pp 62–65

4 W C LYNCH  
A high-speed parsing algorithm for ICOR grammars  
Jennings Computer Center Report 1097 Case Western Reserve

FIGURE 6
DO WHILE COMPILING;
/** ONCE AROUND FOR EACH PRODUCTION (REDUCTION) */

/** LOCATE LEFTMOST REDUCIBLE SUBSTRING
LEFT_END AND RIGHT_END DELIMIT IT */

DO CASE PRODTB(PRD);
/** ONE STATEMENT FOR EACH PRODUCTION OF THE GRAMMAR */

/** <ARITHMETIC TERM> ::= <ARITHMETIC TERM> - <ARITHMETIC PRIMARY> */
DO;
IF TYPE(LEFT_END) = VARIABLE THEN
CALL EMIT(LOAD, LOC(LEFT_END));
CALL EMIT(SUB, LOC(RIGHT_END));
TYPE(LEFT_END) = EXPRESSION;
END;

/** <ARITHMETIC PRIMARY> ::= ARITHMETIC_IDENTIFIER */
DO;
LOC(LEFT_END) = ID_LOOKUP(NAME(LEFT_END));
/** FIND THE LOCATION OF THE IDENTIFIER FROM THE SYMBOL TABLE */
TYPE(LEFT_END) = VARIABLE;
END;

/** <ARITHMETIC PRIMARY> ::= ( <ARITHMETIC EXPRESSION> ) */
DO;
/** SAVE THE INFORMATION IN THE PARALLEL STACKS */
TYPE(LEFT_END) = TYPE(LEFT_END + 1);
LOC(LEFT_END) = LOC(LEFT_END + 1);
END;

/** <ARITHMETIC PRIMARY> ::= # ( <BOOLEAN EXPRESSION> ) */
DO;
/** THE <BOOLEAN EXPRESSION> MUST ALREADY BE IN THE ACCUMULATOR */
CALL EMIT(SHARP_OP, 0); /* UNARY OPERATOR */
TYPE(LEFT_END) = EXPRESSION;
END;

/** ::= */

/** ::= */

END; /* OF CASE ON PRODUCTION NUMBER */

RIGHT_END = LEFT_END;
HEADS(RIGHT_END) = HDTB(PRD); /* THE REDUCTION */

END; /* OF DO WHILE COMPILING */
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>University 1968</td>
<td></td>
</tr>
<tr>
<td>5 W M McKEEMAN</td>
<td></td>
</tr>
<tr>
<td><em>An approach to computer language design</em></td>
<td></td>
</tr>
<tr>
<td>Technical Report CS 48 Computer Science Department</td>
<td></td>
</tr>
<tr>
<td>Stanford University August 1966</td>
<td></td>
</tr>
<tr>
<td>6 W M McKEEMAN J JHORNING D B WORTMAN</td>
<td></td>
</tr>
<tr>
<td><em>A compiler generator implemented on the IBM system/S360</em></td>
<td></td>
</tr>
<tr>
<td>To be published</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 N WIRTH H WEBER</td>
</tr>
<tr>
<td><em>EULER a generalization of algol and its formal definition parts I and II</em></td>
<td></td>
</tr>
<tr>
<td>Comm ACM 9 Jan Feb 1966 pp 13–25</td>
<td></td>
</tr>
<tr>
<td>pp 89–90</td>
<td></td>
</tr>
<tr>
<td>8 N WIRTH C A R HOARE</td>
<td></td>
</tr>
<tr>
<td><em>A contribution to the development of algol</em></td>
<td></td>
</tr>
<tr>
<td>Comm ACM 9 June 1966 pp 413–431</td>
<td></td>
</tr>
</tbody>
</table>
XPL is defined by the BNF grammar given below:

\[
\begin{align*}
\text{<program>} & ::= \text{<statement list>} \\
\text{<statement list>} & ::= \text{<statement>} \\
& \quad | \text{<statement list> <statement>} \\
\text{<statement>} & ::= \text{<basic statement>} \\
& \quad | \text{<if statement>} \\
\text{<basic statement>} & ::= \text{<assignment>} ; \\
& \quad | \text{<group>} ; \\
& \quad | \text{<procedure definition>} ; \\
& \quad | \text{<return statement>} ; \\
& \quad | \text{<call statement>} ; \\
& \quad | \text{<go to statement>} ; \\
& \quad | \text{<declaration statement>} ; \\
& \quad | \text{<label definition>} \text{<basic statement>} \\
\text{<if statement>} & ::= \text{<if clause>} \text{<statement>} \\
& \quad | \text{<if clause>} \text{<true part>} \text{<statement>} \\
& \quad | \text{<label definition>} \text{<if statement>} \\
\text{<if clause>} & ::= \text{IF} \text{<expression> THEN} \\
\text{<true part>} & ::= \text{<basic statement>} \text{ELSE} \\
\text{<group>} & ::= \text{<group head>} \text{<ending>} \\
\text{<group head>} & ::= \text{DO} ; \\
& \quad | \text{DO} \text{<step definition>} ; \\
& \quad | \text{DO} \text{<while clause>} ; \\
& \quad | \text{DO} \text{<case selector>} ; \\
& \quad | \text{<group head>} \text{<statement>} \\
\text{<step definition>} & ::= \text{<variable> <replace> <expression> <iteration control>} \\
\text{<iteration control>} & ::= \text{TO} <expression> \\
& \quad | \text{TO} <expression> \text{BY} <expression> \\
\text{<while clause>} & ::= \text{WHILE} <expression> \\
\text{<case selector>} & ::= \text{CASE} <expression>
\end{align*}
\]
| **<procedure definition>** | ::= | **<procedure head>** <statement list> **<ending>** |
| **<procedure head>** | ::= | **<procedure name>** ; |
| | | **<procedure name>** **<parameter list>** ; |
| **<procedure name>** | ::= | **<label definition>** PROCEDURE |
| **<parameter list>** | ::= | **<parameter head>** <identifier> ) |
| **<parameter head>** | ::= | ( |
| | | **<parameter head>** <identifier> , |
| **<ending>** | ::= | END |
| | | END <identifier> |
| | | **<label definition>** <ending> |
| **<label definition>** | ::= | **<identifier>** : |
| **<return statement>** | ::= | RETURN |
| | | RETURN <expression> |
| **<call statement>** | ::= | CALL <variable> |
| **<go to statement>** | ::= | **<go to>** <variable> |
| **<go to>** | ::= | GO TO |
| | | GOTO |
| **<declaration statement>** | ::= | DECLARE <declaration element> |
| | | **<declaration statement>** , **<declaration element>** |
| **<declaration element>** | ::= | **<type declaration>** |
| | | **<identifier>** LITERALLY <string> |
| **<type declaration>** | ::= | **<identifier type>** |
| | | **<bound head>** <number> ) **<type>** |
| | | **<type declaration>** <initial list> |
| **<type>** | ::= | FIXED |
| | | CHARACTER |
| | | LABEL |
| | | <bit head> <number> ) |
| **<bit head>** | ::= | BIT ( |
| **<bound head>** | ::= | <identifier> ( |
| **<initial list>** | ::= | <initial head> <constant> ) |
| **<initial head>** | ::= | INITIAL |
| | | <initial head> <constant> , |
<assignment> ::= <variable> <replace> <expression> \\
| <left part> <assignment>

<replace> ::= =

<left part> ::= <variable> ,

<expression> ::= <logical factor> \\
| <expression> | <logical factor>

<logical factor> ::= <logical secondary> \\
| <logical factor> & <logical secondary>

<logical secondary> ::= <logical primary> \\
| <logical primary>

<logical primary> ::= <String expression> \\
| <String expression> <relation> <String expression>

<string expression> ::= <arithmetic expression> \\
| <string expression> | <arithmetic expression>

<arithmetic expression> ::= <term> \\
| <arithmetic expression> + <term> \\
| <arithmetic expression> - <term> \\
| + <term> \\
| - <term>

<term> ::= <primary> \\
| <term> * <primary> \\
| <term> / <primary> \\
| <term> MOD <primary>

<primary> ::= <constant> \\
| <variable> \\
| ( <expression> )

<variable> ::= <identifier> \\
| <identifend> ( <expression> )

<subscript head> ::= <identifier> ( <expression> ) \\
| <identifier> ( <expression> ,

<constant> ::= <string> \\
| <number>