XPL to C Source Language Translator

Daniel Weaver

1. Introduction

XPL is a dialect of PL/1 created by W. M. McKeeman et. al. and described in his book A Compiler Generator. XPL was designed to be used in table-driven syntax-directed translators such as the XPL compiler itself. The XPL system, which includes the compiler, SKELETON, and ANALYZER, provides a quick and efficient method for defining a language. If a BNF (BACKUS-NAUR FORM) description of a computer language is input to the ANALYZER, it generates the syntax tables used in either SKELETON or the XPL compiler. The syntax tables along with SKELETON form a syntax checker for your BNF grammar. Code emitters may be added to upgrade the syntax checker to a compiler.

1.1 Source Translation

This implementation of the XPL compiler is an XPL to C source language translator. The compiler translates XPL source to C source which can be compiled on any computer that supports a C compiler. Unlike most XPL compilers this compiler is not a self compiling compiler it is written in C and there is no plan to rewrite it in XPL. The runtime is also written in C. Using C as a target language does cause some limitations on the compiler. These are listed in the section on *Limitations*.

1.2 Language features

The XPL language includes structured program flow, powerful character string manipulators, and fixed and logical operations. Character strings may be from 0 to 2,147,483,647 bytes long. Fixed point operation may be done on 8, 16, 32 or 64 bit quantities. 64 bit integers are supported only if supported by the target C compiler and the underlying hardware.

1.3 Installation

It is my hope that the XPL compiler will run straight out of the box but you still may need to tweak it to get BIT(64) integers to work. If your C compiler does not support **long long** variables then you should change the file **xpl.h** so that the line defining XPL_LONG sets it to just **long**. Like this:

```
#define XPL_LONG long
```

Or you might need to do this:

```
#define XPL_LONG __int64
```

2.0 Constants, Identifiers and Comments

2.1 Identifiers and Constants the Basic Building Blocks of Life

An identifier is an alphabetic (A thru Z, a thru z, #, $, @, and _) optionally followed by any number of alphabetics or numbers (0 thru 9). The first 256 characters of the identifier must be unique. Generally speaking, the compiler loves long names, but identifiers longer than 23 characters will smear the symbol table dump. Keywords in XPL are reserved and may not be used as identifiers. The following symbols are reserved in XPL:

bit	**character**	**end**	**goto**	**literally**	**then**
by	**declare**	**eof**	**if**	**mod**	**to**
call	**do**	**fixed**	**initial**	**procedure**	**while**
case	**else**	**go**	**label**	**return**	**xor**

String constants begin with a single quote and end with a single quote. The single quote itself may be used with a string constant by using two single quotes in a row. Examples of string constants:

```
'This is a string constant'
'Surely, you can''t be serious.'
'I''m very serious.  And stop calling be Shirley.'
```

Integer constants are made up of the digits 0 thru 9. Decimal numbers may not have embedded blanks. Examples:

```
10      42      32768
```

Negative numbers are supported but they are actually expressions.

Binary, octal, and hexadecimal are supported and require the number to be enclosed in double quotes. The number of bits in the radix can be explicitly changed by enclosing a small constant in parentheses. The default radix is 16 so the default number of bits needed for each digit is 4. The radix may be changed within a number by supplying a new base in parentheses. Blanks may be used within bit strings. Macros are not recognized within bit strings. Examples:

```
"(1) 1010 1100"       binary.
"(2) 2230"      quaternary.
"(3) 254"       octal.
"(4) AC"        hexadecimal.
"AC"            also hexadecimal.
"(1) 1 (2) 3 (3) 7 (4) F"       gestalt.
```

All character string and bit string constants are null terminated. Dynamically created strings do not have a null terminator.

As an extension to the XPL language, this compiler allows the base to be any number between 1 and 64 (32 for a 32 bit machine). This allows you to specify hexadecimal numbers that are not aligned to a bit position that is a multiple of 4. When using this feature blanks within strings are significant. For example:

```
"(2)0 (5)1f (6)00 (5)1f (1)0 (13)fff"
```

Defines a SPARC instruction.

Another extension to the original XPL language allows the definition of strings that look similar to strings in C. This feature is activated by using the letter C instead of a number surrounded by parentheses. A C string would be “(c)Hello World!”. This string format recognizes the C escape characters when used in conjunction with the backslash. The following escape characters are supported:

\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage Return
\t	Tab
\v	Vertical Tab
\\	Blackslash
\’	Single Quote
\”	Double quote
\?	Question mark
\xHH	Hex byte
\777	Octal

According to the original XPL definition bit strings longer than 32 bits are converted to **character** type. This had to be modified to allow for 64 bit hexadecimal numbers. Within the compiler the XPL to C source translator keeps bit strings as both integers and character strings and the compiler uses whichever value that requires less type conversions. For example:

declare string character, number bit(32);
string = “(c)HI”;
number = “(c)HI”;

Will set the variable **string** to two bytes ‘H’ and ‘I’. The variable **number** will be set to 18505.

2.2 Comments

The XPL compiler has two types of comments. It supports the // comment as well as the /* */ style comment of PL/I. Comments that start with a // extend to the end of the line and may be placed anywhere on the line. Comments that start with a slash asterisk(/*) extend to the next asterisk slash (*/). Comments may span record boundaries and contain any character. Comments may appear anywhere a blank may appear. Generally comments do not effect the execution of the program and should be used freely to document the text. Example:

/* THIS IS A COMMENT */

Comments may contain control toggles that may have some effect on your program. Control toggles are characters proceeded by a dollar sign ($). The toggles are turned on or off by a dollar sign followed by the character(e.g. $S). Control toggles may also be set on the command line when the compiler is invoked. Most control toggles are upper case letters.

The control toggles may be pushed or popped using $>**<letter>** or $<**<letter>**

respectively.   Example:

```
/* Push the value of H.  $>H  After a push the toggle is OFF */
declare new_variable fixed;
/* Pop the value of H. $<H  The toggle returns to its previous value. */
```

3. Declarations and Data Types

3.1 Data Types

<type> ::= fixed
<type> ::= character
<type> ::= label
<type> ::= <bit head> <expression>)
<type> ::= <character head> <expression>)
<bit head> ::= bit (
<character head> ::= character (

The XPL compiler contains 8 data types that may be used for data storage or arithmetic expressions. Identifiers declared as BIT(1) are assumed to be boolean values that hold only the value of 0 or 1. They are allocated as *unsigned char* and may be used in arithmetic expressions. Identifiers declared as BIT(2) thru BIT(8) are also allocated as *unsigned char* and may hold the values of 0 thru 255. Identifiers declared as BIT(9) thru BIT(16) will be allocated as *signed short* and will be treated as signed 16 bit integers in arithmetic expressions. Identifiers declared as BIT(17) thru BIT(32) or FIXED will be allocated as *signed int* and will be treated as signed 32 bit integers in arithmetic expressions. When supported by the hardware BIT(33) thru BIT(64) will be allocated as *signed long* and treated as 64 bit signed integers. CHARACTER declarations, and BIT declarations larger than 32, (or 64 on 64-bit machines) are considered strings. Strings are not permitted in arithmetic expressions with the exception that you may compare one string to another. Each string has an eight, or twelve, byte string descriptor. This descriptor contains the byte address of the text of the string and four bytes for the string's length.

The fixed length CHARACTER format e.g. CHARACTER(5) is used much like dynamic character strings. All fixed length strings are null terminated. When the string is used as a source the compiler creates a descriptor that holds the address and length of the string. The descriptor format is exactly like the descriptors used for variable length strings. The length of the string is created by finding the null terminator. When a fixed length string is used in a store operation the data in the string is copied to the storage area of the fixed length string and a null terminator is attached to the end of the string. If the string is longer than the space allotted to the fixed length string the string will be truncated and the null terminator will be placed in the last byte of the string.

The LABEL declaration type is used for GOTO's and forward procedure calls.

3.2 Declarations

<declaration statement> ::= declare <declaration element>
<declaration statement> ::= <declaration statement> , <declaration element>
<declaration element> ::= <type declaration>
<declaration element> ::= <identifier> literally <string>
<type declaration> ::= <identifier specification> <type>
<type declaration> ::= <bound head> <expression>) <type>
<type declaration> ::= <type declaration> <initial list>
<identifier specification> ::= <identifier>
<identifier specification> ::= <identifier list> <identifier>)
<identifier list> ::= (
<identifier list> ::= <identifier list> <identifier> ,

If you're gonna use XPL you gotta DECLARE all your variables. The DECLARE statement associates identifiers with various attributes and allocates memory. The following statement will declare I, J, and K to be 32 bit integers.

declare i bit(32), j bit(32), k bit(32);

Data types may be factored as in the following example which also declares I, J, and K to be 32 bit integer.

declare (i, j, k) bit(32);

Declaration statements may be grouped to add readability to the program, or you may group them because you dislike to type the word DECLARE over and over again. The following statement shows a glob of declarations bunched together for aesthetic reasons.

declare any_identifier bit(16), and_receive_a_free character, string bit(8);

When you get tired of declaring scalar variables try arrays. An array is a contiguous block of memory that is referenced by only one identifier followed by a subscript. Arrays in XPL start at subscript zero, therefore when allocating arrays, use one less than the number of elements that you actually need. FORTRAN programmers who think that starting an array reference at zero is inconvenient have three choices. 1) Throw the zeroth element away, 2) Go back to FORTRAN (we didn't want you any way) 3) Learn to count starting from zero. The following example shows an array of HOPE.

declare hope(255) bit(16);

Factoring becomes more complicated with arrays. You must factor both the number of elements and the type. The following example declares three of the seven dwarves to be 32 bit integer arrays of 7 elements each.

declare (dopey, grumpy, sneezy) (6) bit(32);

3.3 Initialization

<initial list> ::= <initial head> <expression>)
<initial head> ::= initial (
<initial head> ::= <initial head> <expression> ,

All static variables are set to zero when the program starts execution unless explicitly initialized by the INITIAL clause of the DECLARE statement. The INITIAL clause sets a value into the data when the program starts. The INITIAL value may be a character string constant or an integer expression.

The INITIAL clause may be used to initialize arrays. When initializing arrays, the first value is stored into the first element of the array (index zero), the second value is placed into the second element and so on. If the number of values is less than the number of elements in the array, the balance of the array is set to zero (or the null string if a CHARACTER array). Example:

```
declare size literally '34',
table(size) bit(16) initial(0, 1, 2, 3, 4, 5, 6, 7),
pt bit(16) initial(size + 1),
name character initial(‘Texas'),
message character(20) initial(' Read this first');
```

3.4 Macro,Definitions

XPL provides a simple text substitution macro. Each time the macro name occurs in the input text the compiler replaces the identifier name with the specified text. This feature is very useful for defining array bounds or abbreviations. The following example shows its usefulness:

```
declare proc literally 'procedure',
        LIMIT literally '127',
        z(LIMIT) bit(16);
sum: proc bit(16);
        declare (i, n) bit(16);
        n = 0;
        do i = 0 to LIMIT;
                n = n + z(i);
        end;
        return n;
end sum;
```

Macros may also be nested, as in the example:

```
declare TRUE literally '1';
declare forever literally 'while TRUE’;
do forever;
        output = ‘Print this forever';
end;
```

In this implementation of the XPL language macros are stored in the symbol table. Therefore they have scope. Any macro declared in a procedure will be undefined when the procedure ends. Macros defined in an outer scope can not be redefined when you enter a new procedure. Use of the macro name in a DECLARE statement will be expanded before the DECLARE statement is processed.

4. Procedures

4.1 Functions and Subroutines

```
<basic statement> ::= <procedure definition> ;
<procedure definition> ::= <procedure head> <statement list>
<ending>
<procedure head> ::= <procedure name> ;
<procedure head> ::= <procedure name> <type> ;
<procedure head> ::= <procedure name> <parameter list> ;
<procedure head> ::= <procedure name> <parameter list> <type> ;
<procedure name> ::= <label definition> procedure
<parameter list> ::= <parameter head> <identifier> )
<parameter head> ::= (
<parameter head> ::= <parameter head> <identifier> ,
<ending> ::= end
<ending> ::= end <identifier>
<ending> ::= <label definition> <ending>
<label definition> ::= <identifier> :
```

The procedure is a clever device designed to facilitate the grouping of mutually confusing statements so that they may be ignored en mass. Procedures that do not return a value are called subroutines. Subroutines must be called with the CALL statement. Procedures that return values are functions. Functions may be referenced as variables or called with the CALL statement. All procedures must be defined or declared before they are referenced. The procedure is defined when the body of the procedure appears in the input text. Procedures need not have any parameters but if any parameters are used they must be DECLAREd before they are referenced within the body of the procedure. Below is an example of a valid subroutine definition:

```
test:
        procedure(a, b, c);
                declare a bit(16), b fixed, c character;
                /* put body of procedure here */
                return;
        end test;
```

Here is the above procedure done as a function:

```
test:
        procedure(a, b, c) bit(16);
                declare a bit(16), b fixed, c character;
                /* put body of procedure here */
                return a + length(c);
        end test;
```

The above procedures use call by value. Call by value moves the value of the calling statement's parameter into the procedure's local storage with type conversion (if necessary). Call by value does not permit arrays to be passed as parameters. When a call by value parameter is changed within the procedure the corresponding parameter in the calling statement is not changed.

Procedures may be DECLAREd using the LABEL type to tell the compiler that the procedure is a forward call. Example:

```
declare test label;
call test(6);
/* other XPL statements */
test:
        procedure(a);
                declare a bit(16);
                /* Add more statements here. */
        end test;
```

The number of parameters in the calling statement must match the number of parameters in the procedure. Type conversion, if possible, will be performed as part of the calling sequence.

4.2 Scope

Identifiers may only be referenced within their proper scope. The scope of an identifier declared before any procedure definition, is any statement following the declaration. Identifiers declared within procedures may only be referenced within the procedure they were declared in. After each procedure ends, all variables defined in the procedure are removed from the symbol table. The storage space allocated to the static variables within the procedure is not deallocated. Values stored in these variables will remain until changed by another call to the procedure. Procedures also permit the redeclaration of variables. The compiler searches the most recent procedure first, then any other nested procedures, then global references, then built-in functions.

This XPL compiler implements nested functions without support for nested functions in the target C compiler. In order to do this most of the variables in the outer function have to be promoted to global scope. This changes where the variables are allocated. Normally they would be allocated on the stack but a nested function will cause them to be static in the generated C code. The XPL compiler will display a list of variables that have been promoted to global scope when using the $D option to dump internal tables at the end of compilation.

4.3 Call and Return

```
<return statement> ::= return
<return statement> ::= return <expression>
<call statement> ::= call <variable>
```

Procedures may be invoked by the CALL statement. When a function is invoked by the CALL statement the value returned by the function is discarded.

Functions may also be referenced as variables within expressions. Example:

```
call scan(text);
token = scan(text);
```

Due to the constraints of the underlying C compiler all functions must use the RETURN statement with a value. All subroutines may only use RETURN statements without a value. If execution of a procedure hits the END statement or a procedure, a return is executed with no value.

4.4 Recursive procedures

XPL does not support recursion. Just because the compiler target is a recursive language does not make the resulting code recursive. Nested procedures cause local variables to be declared static. In order to do garbage collection all character strings are allocated in a single array. Character strings may not be allocated on the stack. This prevents subroutines and functions from being recursive.

5. Control Flow

5.1 IF THEN ELSE

```
<if statement> ::= <if clause> <statement>
<if statement> ::= <if clause> <true part> <statement>
<if statement> ::= <label definition> <if statement>
<if clause> ::= if <expression> then
<true part> ::= <basic statement> else
```

IF statements provide the programmer with a method to conditionally execute statements. The statement following the THEN will be executed if the expression is true. The ELSE statement will be executed, if one exists, if the expression is false. If the expression is a logical variable or an integer the low order bit is tested to determine true or false. Integers are true if the low order bit is one and false if the low order bit is zero. Floating point numbers are converted to integer before testing. Character strings may not be used as logical variables. It is illegal to have two ELSE statements in a row. This handicap may be circumvented by nesting the inner most IF/THEN/ELSE in a DO/END group.

```
i = 1;
if i then output = ‘Always print this';
else output = 'never print this';
if i > 10 then do;
        if j = k then m = 77;
        else m = 66;
end;
else m = 44;
```

Character strings are compared first on length then on the characters in the string. This is usually not what you want. The following procedure will compare strings byte by byte until the end of the shortest string, then length.

```
string_gt:
procedure(l, r) bit(1);
        /* return true if l > r */
        declare (l, r) character,
                (lenl, lenr) bit(32);
        lenl = length(l);
        lenr = length(r);
        if lenl <= lenr then return l > substr(r, 0, lenl);
        else return substr(l, 0, lenr) >= r;
end string_gt;
/* The following will reference the function */
if string_gt(a, b) then /* do something */
```

5.2 DO CASE

```
<group> ::= <group head> <ending>
<group head> ::= do <case selector> ;
<group head> ::= <group head> <statement>
<case selector> ::= case <expression>
```

The DO CASE statement uses the expression to select the Nth statement within the body of the DO group. After the statement is executed control is returned to the CASE's END statement. Statements are numbered from zero starting from the first statement after the DO CASE. If the expression is negative or larger than the number of statements in the DO CASE, a random jump will be executed (Crash City). Below is an example of a valid CASE statement:

```
declare i bit(16);
i = 1;
do case i;
        output = 'never print this';
        output = 'always print this';
        output = 'don''t print this';
end;
output = ‘This is executed next';
```

5.3 DO WHILE

```
<group> ::= <group head> <ending>
<group head> ::= do <while clause> ;
<group head> ::= <group head> <statement>
```

<while clause> ::= while <expression>

The DO WHILE statement will loop until the expression becomes false. The expression is evaluated before the loop is entered. If the expression is false the body of the DO WHILE is never executed. Integers are true if the low order bit is one and false if the low order bit is zero. If the expression is false control is passed to the statement after the END statement. Below is an example of a valid DO WHILE statement.

```
declare i bit(16);
i = 0;
do while i < 4;
        i = i + 1;
        output = 'testing ' || i;
end;
/* the above loop will be executed 4 times */
```

5.4 DO Loops

<group> ::= <group head> <ending>
<group head> ::= do <step definition> ;
<group head> ::= <group head> <statement>
<step definition> ::= <variable> <replace> <expression> <iteration control>
<iteration control> ::= to <expression>
<iteration control> ::= to <expression> by <expression>

It is a little known fact that the DO loop was invented by a file clerk who worked for a small computer manufacturer at Hursley Laboratory in the UK. The programmers in this computer company spent much of their time in committee designing large compiler projects. The statements made in each meeting had to be numbered, filed and sent back to committee for review. The file clerk, whose name is David Owens, was given the task of filing the results of the meetings. David initialed each report before sending it back to the committee. The committee came to refer to the iterative processing of statements as the David Owens loop, or DO loop for short.

The DO loop provides iterative execution of a group of statements a certain number of times. The DO loop provides a start value, and end value and an optional increment. When the DO loop begins it sets the iteration variable to the start value. If the iteration variable exceeds the end value then the body of the DO loop is never executed. Otherwise the statements within the DO loop are executed. After executing the statements in the loop, the variable is incremented by the expression in the BY clause of the DO statement. Then the whole process repeats. If the BY clause is omitted an increment of one is used. Negative constants are permitted in the DO loop BY clause but all non-constant expressions are considered positive. The DO loop variable may not be indexed.

The escape value and loop increment can not be changed from within the DO loop, but the DO loop variable may be modified by the statements within the loop. Here are some examples:

```
declare c character, k fixed;
c = 'testing ';
do k = 1 to 3;
        c = c ll k ll ' ';
end;
/* At this point c = 'testing 1 2 3 ' */
declare x, y, z;
y = 10;
z = 20;
do x = y to z by 2;
        call it;
        y, z = 4;
end;
/* The above loop is executed 6 times x=10 x=12 x=14 x=16 x=18 x=20
*/
```

5.5 GO TO

```
<basic statement> ::= <go to statement> ;
<go to statement> ::= <go to> <identifier>
<go to> ::= go to
<go to> ::= goto
```

The GOTO statement should be used only under extreme conditions, such as "My computer is on fire." Indiscriminate use of the GOTO will brand you forever as an unstructured programmer and cause you to be deleted from my Christmas card list. In the off chance that your computer is on fire, the following rules must be observed; The label must be DECLAREd if it is to be used as an argument to the ADDR function before it is referenced. The label must be DECLAREd if referenced within a procedure before it is defined and that same identifier is declared outside of the procedure. GOTO statements may not jump into or out of a procedure. Indexes are not permitted on GOTO statements.

6. Expressions

6.1 Operator Precedence

Operator precedence is formally defined by the BNF description of the language. For those of you who couldn't be bothered, here is the same information in table form.

*	1	Multiply.
/	1	Divide.

MOD	1	Modulo. The result may be positive or negative.
+	2	Add.
+	2	Unary add (does nothing).
-	2	Subtract.
-	2	Unary subtract.
||	3	String concatenate. If the operands are not character strings they will be converted to character strings. The operation will probably result in the movement of one or both of the operands in memory.
=	4	Equal.
~ =	4	Not equal.
>	4	Greater than.
<	4	Less than.
> =	4	Greater than or equal to.
~ <	4	Not less than. Same as greater than or equal to.
< =	4	Less than or equal to.
~ >	4	Not greater than. Same as less than or equal to.
~	5	Logical not. Note that this is not the priority you normally expect logical not. Normally it should be at priority 2. This may cause confusion when used with compare operators.
&	6	Logical AND.
|	7	Logical OR.
XOR	7	Exclusive OR.

Operators at the same priority will be evaluated left to right.

6.2 Type Conversion

Expressions are evaluated as BIT(8), BIT(16), BIT(32), and BIT(64). Before any arithmetic operation is done the two operands must be of the same type. The smaller precision operand will be converted automatically to the larger precision. Conversion to CHARACTER will be done only when using the concatenate operator, the assignment statement, or certain built-in functions. Fixed length character strings have the same rules as variable length character strings.

Internally the compiler represents all integer constants as the largest precision supported by the C compiler, BIT(32) or BIT(64).

Condition -> Integer	If TRUE then 1, else 0.
Bit(8) -> Condition	If ZERO then FALSE, else TRUE.
BIT(8 thru 64) -> Condition	If the least significant bit is set then TRUE, else FALSE
Integer -> CHARACTER	The integer is converted to a signed decimal character string.
CHARACTER -> Integer	Illegal
CHARACTER -> CHARACTER(<number>)	The string is copied and a null terminator is added.

CHARACTER(<number>) -> CHARACTER A descriptor is created. The length is determined by the position of the first null.

7. The Run-time package

7.1 I/O Considerations

All I/O operations in XPL require a unit number. The runtime package has a table that maps unit number to FILE pointer. The same table is used for both input and output except for unit zero. The default unit numbers are:

Input unit 0 -> stdin	Output unit 0 -> stdout
Unit 1 -> stderr	

All other unit numbers are undefined when the program starts up. They may be assigned by using the builtin function xfopen().

I/O errors will set the variable **errno** to the value of the errno variable which is normally set by the Operating System.

7.2 The Compactify Procedure

COMPACTIFY is the procedure in the run-time package that garbage collects the free string area. When an XPL program starts it calls MALLOC() to get space for the free string area. The amount of space requested can be modified by declaring a macro for the identifier FREESPACE in the main body of the program. The value should be an integer in a format acceptable to the C programming language. The value is the number of bytes requested. Example:

declare FREESPACE literally ‘0x10000’;

Once the free string area is allocated there is no ability to expand it.

The COMPACTIFY procedure uses the variable, **space_needed**, to determine how much space a particular operation will need to complete. Some operations such as input default to 1024 bytes. If COMPACTIFY cannot satisfy the amount of space needed then the program will abort with the following message printed on stderr:

***** Notice from compactify(): Insufficient string space. Job abandoned. *****

The garbage collection process has two forms. The first is a major collection where the entire string space is collected. After a major collection the variable **lower_bound** is set to the top if the free string area. The second type of collection is where it only collects strings above the **lower_bound**. This is call a minor collection. The hope is that the minor collections will free up enough space so that a major collection is not needed.

When searching for strings to collect the COMPACTIFY procedure will ignore any string that does not fall inside the free string area. Fixed length strings are never garbage collected.

8. Limitations and Restrictions

8.1 Limitations and missing features

Using C as a target language enforces a number of restrictions on the XPL compiler. Most of these are minor but I list them here so you can decide. In the following text XPL69 refers to the original XPL implementation as described in A Compiler Generator.

1) When calling a function or subroutine you should supply all arguments defined by the procedure. XPL69 allowed arguments to be omitted and they would retain values from the previous call. Since the C compiler requires the number of arguments to match the XPL compiler adds arguments to fill out the procedure call. These arguments are set to zero.

2) Declare statements may only be used at the beginning of a block. This exactly matches the restrictions in the C language. XPL69 allowed declarations to be anywhere.

3) Functions can not assume a default type. The type of a function must be explicitly declared.

4) Return statements must match the declaration type of the procedure. Functions must only use RETURN <expression> and subroutines must only use RETURN without <expression>

5) Formal parameters may not be arrays. This restriction may be lifted in the future. The implementation would pass only the value at index zero.

6) GOTO statements cannot enter or exit procedures. I see this as an improvement.

7) XPL69 allows control to reach the end of a non-void function. In the original compiler it returned a random value. This XPL compiler will return zero.

8) String variables are stored in READ-ONLY memory. When using the C compiler on a MAC you can suppress this with the -fwritable-strings option.

9) Some variables declared within a procedure may not retain their values after the procedure exits. XPL69 made all variables **static**. This allowed their values to be retained across procedure calls. This XPL compiler puts some variables on the stack which makes them lose their value when the procedure exits.

10) Variables that are not declared as arrays may not have a subscript. XPL69 allowed all variables to be subscripted. For example the following is illegal:

declare a fixed, b(20) fixed;
a(3) = 2;   /* This is ILLEGAL */

11) The order of variables in memory may not resemble XPL69. The original XPL compiled used the order of declaration to simulate overlapping memory something similar to the UNION declaration in C. The following is legal but probably won't work:
declare a(0) fixed, b(255) bit(8);
a(3) = 2;

Appendix A --- Compiler Options ---

```
Usage:
  xcom [-DGILmMPSTUWX] [-v number] [-o outfile] [-s stringfile] file
    -o outfile - Output file name for C code
    -s stringfile - Output string header file name
    -v number - Number of entries reserved for the argv array
    file - XPL input source file name

  Uppercase letters set initial values for compiler toggles
    D - Dump stats at the end of compilation
    G - 32 bit machine (2 Gig address space)
    H - Hide identifiers from the C compiler
     I - Ignore case for keywords and builtin functions
    K - Emit Linemarkers for the C compiler
    L - List source
    m - Do not generate code for main()
    M - Minimal source listing; takes precedence over -L
    S - Dump symbol table at end of each procedure
    T - Trace compiler productions
    U - Give warning for unused variables
    W - Inhibit warnings for undeclared variables and procedures
    X - Show macro expansions
    Y - Give warning for high order truncation
```

-v number	Set the number of entries reserved for the argv array in the compiled program. 32 is the default. If your program needs more than 32 command line options you may use this option to expand the number of descriptors used in the argv array. This option has no effect on the command line of the compiler itself.
-o outfile	Sets the name of the C source file that will be passed to the C compiler. The default is the basename of the input file with a .c appended.
-s stringfile	Set the name of the C header file that will be used by the C

	compiler. The default is the basename of the object file with a .xh extension.
file	Is the name of the XPL source file. I suggest using .xpl as an extension. If missing the compiler will read input from stdin. When the source file name is missing the -o option must be used.

Compiler Toggles may be set on the command line or embedded in PL/I style comments.

-D	Dump stats at the end of compilation.
-G	Use this option when cross compiling from a 64 bit machine to a 32 bit machine. This option turns off BIT(64).
-H	Hide identifiers from the C compiler. If your program uses an identifier name that is a standard C function such as strlen() or memcpy() you can use this option to tell the XPL compiler to mangle the names so they do not conflict with default definitions in the C language. When this toggle is used within a comment it only needs to be placed around the definition of the variable. Example:

```
/* $H  Hide this variable from the C compiler */
declare strlen fixed;
/* $H  Turn off the control toggle */
```

-I	Ignore case. This allows the use of uppercase letters for keywords and builtin functions.
-K	Emit Linemarkers for the C compiler. This tells the compiler to generate #line directives when emitting code that is passed to the C compiler. This should help you find errors that show up when the C source is compiled but were not detected by the XPL compiler.
-L	List source. Back in the day, programmers would use line printers to get listing of their programs. The listings would be annotated with line numbers and procedure names and the like. You can capture this listing by redirecting stdout to a file.
-m	The code will be generated without a main() procedure. This will allow some other program written in a different language to call the XPL program.
-M	Minimal source listing. List the source code without all the useful annotations. This takes precedence over -L.
-S	Print a symbol table dump at the end of the program and the end of each procedure on stdout.
-T	Trace compiler productions. This is useful for debugging the compiler to tell when a line of text is being created for output. This is way too noisy to be generally useful.
-U	Give warnings for unused variables.
-W	Inhibit warnings for undeclared variables and procedures.
-X	Show macro expansion. When a macro defined by the LITERALLY statement is expanded the compiler will show the expansion and print a line to stdout.

-Y	Give warning for high order truncation. When this option is set a warning is printed when the program stores tries to store a large value into a smaller memory location.

Appendix B --- Built-in Functions ---

__LINE__	A macro the expands to the current line number of the input text.
abort	A subroutine that will abort execution of the current program.
addr(v)	A builtin function that returns the address of the variable v with respect to subscripts.
address	A macro that expands to BIT(32) on 32-bit machines and expands to BIT(64) on 64-bit machines. This is a transparent way to hold an address when porting between machines with different size addresses.
argc	A 32-bit integer that holds the number of elements in the argv array. Note: This holds the number of elements exactly like the argc variable in C. It does not hold the number of elements minus one as is the standard practice in XPL.
argv(31)	A CHARACTER string array the holds the values of the options passed on the command line. The maximum number of elements in the argv array can be changed with the -v option at compile time.

build_descriptor(length, address) Build a string descriptor with the **length** and **address**.

byte(string, position) This is a builtin function that acts like a preudo-array which allows access to the string at the selected position. The position starts counting at zero. The byte function may appear on either the left or the right of the assignment operator. The sequence:

```
        S = ‘ABC’;  I = byte(S, 1);
```

Returns the value of ‘B’ (“42”). The following is an example of byte() used on the left of an assignment operator:

```
        S = ‘ABC’;  S = S II S;  byte(S, 2) = “41”;
```

The above sequence will assign the value of ‘ABAABC’ to S. The byte() function modifies the string but does not move it. Any SUBSTRings pointing to the same area will also be modified. The byte() function can be used with both dynamic CHARACTER strings as well as fixed length strings.

byte(string)	Short form of the above function which refers to the first element of the string. This is equivalent to byte(string, 0).

compactify A subroutine used for garbage collection of the string area. The casual user need never call it, but it is useful if you wish to control the exact point at which garbage collection is done.

corebyte(offset) A BIT(8) array that starts at location zero and allows access to the entire memory. This array may be used on either side of the assignment operator.

corehalfword(offset) A BIT(16) array that starts at location zero and allows access to the entire memory. This array may be used on either side of the assignment operator.

coreword(offset) A BIT(32) array that starts at location zero and allows access to the entire memory. This array may be used on either side of the assignment operator.

date A function that returns today's date as a 32-bit integer in the format:

(year - 1900) * 1000 + day_of_year

date_of_generation A bit(32) variable the holds the value of the date function when the code was compiled.

descriptor() An array which holds all the descriptors used by the program.

errno A BIT(32) variable the will be set to the value of the Unix/Linux errno variable if there is an I/O error in the XPL runtime. This variable is set to zero if the most recent I/O operation was successful. The XPL variable **errno** is not a direct mapping on the C language variable **errno**. This variable only gets set when an error occurs in the XPL runtime.

exit(v) A function that causes the program to exit with a value v. XPL69 implements exit() as an abnormal exit that causes a core dump. I have chosen to implement the C style of exit that causes a normal termination of the program.

expand_tabs(string, tabstop) A CHARACTER function that expands the tabs in the **string** using **tabstop** as the tab position. This function returns the new string.

file(unit, position) A builtin function that provides random access to the I/O device associated with unit. The length of the operation is taken from the size of the array being read or written. **position** is in multiples of the buffer size. **position** start at zero. **file** is used like an assignment statement and the other side of the assignment statement is an array that is used to hold the data. Example:

declare buf(1023) bit(8), (unit, p) fixed;
p = 0;
buf = file(unit, p); /* Read 1024 bytes of record zero into the array buf */
file(unit, p + 1) = buf; /* Write the data back to the next record. */

The file function call may not be used on both the left and right of the assignment in the same statement.

I hate this function. It's original design used a record length that was hard coded in the runtime. The maximum record length may be changed with the **file_record_size** variable.

file_record_size A BIT(64) (or BIT(32)) variable the holds the maximum record size for the **file** builtin function. This may be used to read or write less than the buffer size used in the **file** statement. If set to zero this value is ignored. XPL69 sets the record size in an assembly time constant when building the submonitor. If you have an IBM 2311 disk set this value to 3600. If you have an IBM 2314 disk set this value to 7200.

freebase A BIT(64) (or BIT(32)) variable that points to the byte address of the start of the free string area. Don't play with it.

freelimit A BIT(64) (or BIT(32)) variable that points to the byte address of the and of the free string area. Don't play with this on either.

freepoint A BIT(64) (or BIT(32)) variable that points to the byte address of the next available byte of the free string area. When **freepoint** exceeds **freelimit** the **compactify** procedure is automatically called to compress the free string area. Just so long as you don't go mucking about with my pointers.

hex(value) This builtin function returns the hexadecimal equivalent of the **value** as a character string.

inline(string, …) This builtin function will insert a string into the C code being generated by the compiler. Examples:

call inline('#define MAXNAME 7');
call inline('extern char * user_string;');

Note that the semi colon is not supplied by the compiler. The above statements will generate the following code.

```
#define MAXNAME 7
        extern char * user_string;
```

Inline may also be used as a function on either the left or the right of the assignment operator. Example:

```
inline('special_value') = inline('i ^ 7');
```

The above statement will generate the following code:

```
special_value = i ^ 7;
```

When used as a function the compiler supplies a semi colon.

input(unit) A CHARACTER function that reads one record from the specified **unit**. Unit zero defaults to **stdin**. Example:

```
declare text character;
text = input(2);       /* Read one string from unit 2 */
```

The input function reads characters until it finds a newline (hex code “0A”). Carriage returns are discarded. If you open a file as binary it will return all characters in the input stream. Reading a binary file will typically return a string of 1024 bytes. See **xfopen()** function.

input Same as input(0)

length(string) A function that will return the length of a character string. See also **saddr()**.

ndescript A BIT(32) variable the holds the number of descriptors minus one. This value should never be changed.

output(unit) A CHARACTER pseudo-variable that has the effect of writing a string to the specified **unit**. Unit zero is **stdout**. Unit one is **stderr**. A newline is appended to each output line unless the output is opened as a binary device with **xfopen()**. Examples:

```
output(0) = 'Hello World!';         /* You were expecting this.
        Right? */
output(1) = 'Major error detected.';
```

I have chosen to follow the C convention rather than the XPL69 standard. Seriously, when was the last time you had a line printer connected to your computer?

output Same as **output(0)**.

saddr(string) This function returns the address portion of the string descriptor. If this is a dynamic string the address could be changed by the compactly subroutine. As a result the address has a very limited lifespan. The address field of a null string may contain garbage. You should check for length equals zero before using this function.

shl(value, count) A logical left shift of the **value**, **count** bits. Left shifting a constant a constant number of bits is done at maximum precision within the XPL compiler. The behavior of this function when doing type conversions and handling weird shift counts is left up to the underlying C compiler. Example:

```
j = 2;
k = shl(j, 2);  /* k will be set to 8 */
```

shr(value, count) A logical right shift of the **value**, **count** bits. This shift does a zero fill on the most significant bits of **value**. Right shifting a constant a constant number of bits is done at maximum precision within the XPL compiler. The behavior of this function when doing type conversions and handling weird shift counts is left up to the underlying C compiler. Example:

```
j = 8;
k = shr(j, 2);  /* k will be set to 2 */
```

Right shifting negative constants will probably give you an unexpected result.

substr(string, start, length) This CHARACTER function permits the dissection of strings. The **start** is character position counting from zero. The **length** will be the number of characters in the resultant string. This function will not create a new string that lies outside the original string. If **start** is larger than the length of the original string this function will return a null string. Only a new descriptor is created the data in the string is not moved. This function may only be used on the right hand side of the assignment operator.

substr(string, start) This form of the **substr** function returns the balance of the string starting at position **start**. It is shorthand for:

substr(string, start, length(string) - start)

This function may only be used on the right hand side of the assignment operator.
Only a new descriptor is created the data in the string is not moved.

time A BIT(32) function that returns the time-of-day in centiseconds since midnight.

time_of_generation A BIT(32) variable with value of **time** when the program was compiled.

trace This subroutine does nothing. It is included here for compatibility.

unique(string) A CHARACTER function that moves the string to the top of the free string area and returns a new descriptor. The function guarantees that the string has only one descriptor pointing to it. Modifying this string with the **byte()** function will have no side effects. The original XCOM compiler uses **SUBSTR(' ' || ' ', 1)** to generate a unique string. This will not work with this compiler. This compiler will concatenate the two constant strings creating one string of two bytes. The following code does work on this compiler and has the same result as the **unique()** function:

```
string = ' ';
unique_string = substr(string || string, 1);
```

untrace This subroutine does nothing. It is included here for compatibility.

xfclose(unit) A subroutine that interfaces to the C **fclose()** function. The **unit** number should be a number returned by **xfopen()**.

xfopen(filename, mode) A function that interfaces to the C **fopen()** function. This function returns a **unit** number if successful and returns -1 if there was an error. The **filename** is an XPL CHARACTER string that holds the name of the file to be opened. The **mode** is one of the following:

'r' Open text file for reading. The stream is positioned at the beginning of the file.

'r+' Open for reading and writing. The stream is positioned at the beginning of the file.

'w' Truncate to zero length or create text file for writing. The stream is positioned at the beginning of the file.

'w+' Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

'a' Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file, irrespective of any intervening fseek(3) or similar.

'a+' Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file, irrespective of any intervening fseek(3) or similar.

The **mode** can also include the letter ‘b’. This will cause the input function to return the data exactly as read from the device and the output function will not append a newline.

A successful open will allow both reads and writes to the **unit**. It is up to the programmer to prevent reads to write only devices and writes to read only devices.

Example:

```
unit = xfopen('/tmp/sourcefile', 'w');
if unit < 0 then do;
        output(1) = 'File open error.  Program aborted.';
        call exit(1);
end;
```

xfprintf(unit, format, args, …) A print function that works similar to the fprintf function in the C programming language. This function returns the number of characters printed. The unit may be opened with the xfopen function call. Example:

```
declare age fixed initial(4);

fprintf(unit, "(c) I am %d years old today.\n", age);
```

xprintf(format, args, …) A print function that works similar to the printf function in the C programming language. This function returns the number of characters printed. Example:

```
declare age fixed initial(4);

printf("(c) I am %d years old today.\n", age);
```

xsprintf(buffer, format, args, …) A character conversion function that works similar to the sprintf function in the C programming language. The buffer may be a dynamic string or a fixed length string. The buffer may not be an expression. The function returns the number of characters put into the buffer. Example:

```
declare age fixed initial(4);
declare string character;
declare text character(80);

xsprintf(string, 'Help me %s.  Your my only hope.',  'Obi Wan');
```

xsprintf(text, ‘string=“%s”, age=%3.2d’, string, age);

/* The buffer is cleared at the beginning of the function call. */

Appendix C --- Format strings ---

The functions xfprintf, xprintf, and xsprintf all use the same format strings. The general form of a format string is:

%[flags][width][.precision][length]specifier

The following flags are supported:

+	Positive signed integers should be proceeded by a plus sign.
-	Left justify
<sp>	Space. Numbers should be proceeded by a space. Ignored if + is used.
#	Prefix octal, hex (lower case) and hex (upper case) with 0, 0x and 0X respectively.
0	Zero fill. Ignored if left justify.

width — Is the field width. This is a decimal number or a *. If a * then the next argument is used as the width. The argument should be a BIT(32) value. If the width is smaller than the converted string value then the width is ignored.

precision — For integer values this is the minimum number of digits to be printed. For strings this is the maximum length of the string. This field may be a decimal number or a *. If a * then the next argument is used as the precision. The argument should be a BIT(32) value.

length — One of the following:

h - BIT(16)
hh - BIT(8)
l, ll, j, z, t, L - BIT(32) or BIT(64) depending on your hardware.

If the length is missing the function will assume BIT(32).

specifier — One of the following:

%	Print %
d, i	Integer
u	Unsigned integer
o	Octal
x	Hexadecimal with lower case letters
X	Hexadecimal with upper case letters
s	Character string

p	Address as returned by the ADDR() function
c	A single character
n	Returns the number of characters written so far. The argument must be an address that points to a BIT(32) integer.

Appendix D --- BNF ---

```
<program> ::= <statement list>
<statement list> ::= <statement>
<statement list> ::= <statement list> <statement>
<statement> ::= <basic statement>
<statement> ::= <if statement>
<basic statement> ::= <assignment> ;
<basic statement> ::= <group> ;
<basic statement> ::= <procedure definition> ;
<basic statement> ::= <return statement> ;
<basic statement> ::= <call statement> ;
<basic statement> ::= <go to statement> ;
<basic statement> ::= <declaration statement> ;
<basic statement> ::= ;
<basic statement> ::= <label definition> <basic statement>
<if statement> ::= <if clause> <statement>
<if statement> ::= <if clause> <true part> <statement>
<if statement> ::= <label definition> <if statement>
<if clause> ::= if <expression> then
<true part> ::= <basic statement> else
<group> ::= <group head> <ending>
<group head> ::= do ;
<group head> ::= do <step definition> ;
<group head> ::= do <while clause> ;
<group head> ::= do <case selector> ;
<group head> ::= <group head> <statement>
<step definition> ::= <variable> <replace> <expression> <iteration control>
<iteration control> ::= to <expression>
<iteration control> ::= to <expression> by <expression>
<while clause> ::= while <expression>
<case selector> ::= case <expression>
<procedure definition> ::= <procedure head> <statement list> <ending>
<procedure head> ::= <procedure name> ;
<procedure head> ::= <procedure name> <type> ;
<procedure head> ::= <procedure name> <parameter list> ;
<procedure head> ::= <procedure name> <parameter list> <type> ;
<procedure name> ::= <label definition> procedure
<parameter list> ::= <parameter head> <identifier> )
<parameter head> ::= (
<parameter head> ::= <parameter head> <identifier> ,
<ending> ::= end
```

```
<ending> ::= end <identifier>
<ending> ::= <label definition> <ending>
<label definition> ::= <identifier> :
<return statement> ::= return
<return statement> ::= return <expression>
<call statement> ::= call <variable>
<go to statement> ::= <go to> <identifier>
<go to> ::= go to
<go to> ::= goto
<declaration statement> ::= declare <declaration element>
<declaration statement> ::= <declaration statement> , <declaration
element>
<declaration element> ::= <type declaration>
<declaration element> ::= <identifier> literally <string>
<type declaration> ::= <identifier specification> <type>
<type declaration> ::= <bound head> <expression> ) <type>
<type declaration> ::= <type declaration> <initial list>
<type> ::= fixed
<type> ::= character
<type> ::= label
<type> ::= <bit head> <expression> )
<type> ::= <character head> <expression> )
<bit head> ::= bit (
<character head> ::= character (
<bound head> ::= <identifier specification> (
<identifier specification> ::= <identifier>
<identifier specification> ::= <identifier list> <identifier> )
<identifier list> ::= (
<identifier list> ::= <identifier list> <identifier> ,
<initial list> ::= <initial head> <expression> )
<initial head> ::= initial (
<initial head> ::= <initial head> <expression> ,
<assignment> ::= <variable> <replace> <expression>
<assignment> ::= <left part> <assignment>
<replace> ::= =
<left part> ::= <variable> ,
<expression> ::= <logical factor>
<expression> ::= <expression> I <logical factor>
<expression> ::= <expression> xor <logical factor>
<logical factor> ::= <logical secondary>
<logical factor> ::= <logical factor> & <logical secondary>
<logical secondary> ::= <logical primary>
<logical secondary> ::= ~ <logical primary>
<logical primary> ::= <string expression>
<logical primary> ::= <string expression> <relation> <string expression>
<relation> ::= =
<relation> ::= <
```

```
<relation> ::= >
<relation> ::= ~ =
<relation> ::= ~ <
<relation> ::= ~ >
<relation> ::= < =
<relation> ::= > =
<string expression> ::= <arithmetic expression>
<string expression> ::= <string expression> || <arithmetic expression>
<arithmetic expression> ::= <term>
<arithmetic expression> ::= <arithmetic expression> + <term>
<arithmetic expression> ::= <arithmetic expression> - <term>
<arithmetic expression> ::= + <term>
<arithmetic expression> ::= - <term>
<term> ::= <primary>
<term> ::= <term> * <primary>
<term> ::= <term> / <primary>
<term> ::= <term> mod <primary>
<primary> ::= <constant>
<primary> ::= <variable>
<primary> ::= ( <expression> )
<variable> ::= <identifier>
<variable> ::= <subscript head> <expression> )
<subscript head> ::= <identifier> (
<subscript head> ::= <subscript head> <expression> ,
<constant> ::= <string>
<constant> ::= <number>
```