A PROGRAMMABLE FORTRAN PREPROCESSOR

M. Rosing

June 1995

Presented at the
ACM Sigplan Conference on Programming Language
Design & Implementation
June 18-21, 1995
LaJolla, California

Prepared for
the U.S. Department of Energy
under Contract DE-AC06-76RLO 1830

Pacific Northwest Laboratory
Richland, Washington 99352

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
A Programmable Fortran Preprocessor
Matt Rosing
Pacific Northwest Laboratory
m_rosing@ pnl.gov, 509 372-4467

Abstract
A programmable Fortran preprocessor is described. It allows users to define compile
time operations that can examine and modify the source tree before it is compiled with a
traditional compiler. This intermediate step allows the definition of routines and operations
that adapt to the context in which they are used. Context sensitive operations increase
the flexibility of abstractions that can be built without degrading efficiency, as compared to
using traditional run time based abstractions such as libraries or objects.

The preprocessor is described briefly along with an example of how it is used to add
CMFortran array operations to Fortran77. Other preprocessors that have been implemented
are also briefly described.

Keywords: Extensible language, Compilers, Preprocessor, High level languages,
Scientific computing

1 Introduction.

A programmable preprocessor to support domain specific compilers is described. The prepro-
cessor, based on Fortran, allows a user to define compile time operations on program segments
for supporting efficient, low level abstractions. A programmable preprocessor has two unique
features that, when combined, make it practical to develop more general abstractions than is
currently possible using methods such as libraries or object oriented technology. The first is
that the abstractions are executed at compile time, resulting in a modified source program.
This permits context sensitive execution of abstractions and allows computation, in the form
of generating code specific to a problem, to be done at compile time as opposed to run time.
The result is more efficient run time code given fairly general abstractions, when compared
to libraries or object oriented techniques. The second unique feature of a programmable pre-
processor is that it is fairly simple to program. The parse tree is treated as a single, large
expression and the user overloads operators, functions, objects, and grammar rules. In the
process of executing a node in the parse tree, a function can examine or modify any part of
the tree. In this manner, the parse tree is also treated as a data structure. All other details of
the transformation process are handled by the underlying system. This includes, for example,
parsing, name and type analysis, and code generation.

An example of a source program using compile time abstractions is shown in Figure 1. This
code is one line of a CMFortran program that was ported to a KSR-2. The code models the
transport of pollutants through the ground and each of fh, c, and qx are arrays. The cshift
operator shifts an index, specified by the second parameter, an amount specified by the third
parameter. The resulting code, using POSIX threads executing on a globally addressed machine
is shown in Figure 2. In this case the data is distributed by panels and the computation of
\[ \text{fh} = c \times 0.5 \times (\text{qx} + \text{cshift}(\text{qx}, 1, -1)) \]

Figure 1: A CM-Fortran Array Operation.

```
DO kx=13_13,h3_14
DO jx=12_11,h2_12
DO ix=11_9,h1_10
    fh(ix,jx,kx) = c(ix,jx,kx) \times 0.5 \times (\text{qx}(ix,jx,kx) + \text{qx}(ix+(-1),jx,kx))
ENDDO
ENDDO
ENDDO

CALL synch ! synch with neighbors before doing communication
update borders

IF(me.GT.1)THEN
    CALL move_border(fh,11_9,h1_10,8+4,13_13,h3_14,1+4,(8/2+4)*2,12_11-2,(12_11-2)-4)
    CALL move_border(fh,11_9,h1_10,8+4,13_13,h3_14,1+4,(8/2+4)*2,12_11-1,(12_11-1)-4)
ENDIF

IF(me.LT.2)THEN
    CALL move_border(fh,11_9,h1_10,8+4,13_13,h3_14,1+4,(8/2+4)*2,h2_12+2,(h2_12+2)+4)
    CALL move_border(fh,11_9,h1_10,8+4,13_13,h3_14,1+4,(8/2+4)*2,h2_12+1,(h2_12+1)+4)
ENDIF

CALL synch ! synch with neighbors before doing next operation
```

Figure 2: Translated Code of Figure 1, Executable Using Posix Threads
any array operation is broken into two parts. The first does the local computation and the second updates the edge elements. This example will be described in more detail in Section 2 but it is important to note the complexity of the process that generates the resulting code. Information is collected from the declaration of distributed arrays, the form of the expression, and the context of the statement. The distributed arrays define the number and ranges of do loops. The expression defines the body of the do loops. Finally, the context is important because, where the statement is within a where clause the processing is substantially different.

The crux of the solution is to allow the user to write functions that modify the program at compile time. To manage the complexity, the program is treated as both a single, large expression, and at the same time, a tree based data structure. This combined object is called a "parse tree expression." Each node of the tree corresponds to a grammar rule in the base language. The user defines functions that operate on the grammar rules and return new code segments. Furthermore, these functions can modify other areas of the parse tree expression, such as adding declarations or code to initialize data structures. The execution of the parse tree expression results in a new parse tree corresponding to a Fortran program that is compiled with a standard compiler to generate the object file.

By limiting the complexity of the preprocessor in this manner, the source code that defines the sample preprocessor is under 3000 lines of code. This preprocessor is considerably more powerful than just translating array operations. For example, it has support for defining and using dynamic arrays, user defined data distributions, and user defined iterators on the local part of distributed arrays. This makes it fairly easy to port CMFortran codes from, a KSR, for example, to a Paragon. The semantics of this preprocessor is beyond the scope of this paper and is the subject of another one concurrently being written. In comparison to the preprocessor, however, another one that was developed to expand Fortran90-like array operations using triplet notation is approximately 800 lines of code.

In contrast, to build a set of abstractions using macros, libraries, or object oriented techniques that support this functionality and efficiency, would be extremely difficult if not impossible. Another approach would be to develop a compiler. While this could obviously support the functionality and efficiency required, it would be considerably larger and therefore would probably not be built. As an example, in the context of porting CMFortran programs to a KSR, we were willing to spend a few weeks of time developing a preprocessor. But it was not worth the effort to build an entire compiler as such a compiler would have been on the order of 15,000 lines long. Furthermore, the knowledge required to build a compiler is much larger than that of building a preprocessor. Thus, the programmable preprocessor has filled a niche that can not be filled by other, existing technologies.

Section 2 describes the example in more detail. Section 3 compares the use of a programmable preprocessor to solve this problem with that of other technologies. Section 4 describes the basic structure of the preprocessor and Section 5 is a brief conclusion.
2 Example Translation.

This section describes in more detail the process of translating the code of Figure 1 into that of Figure 2. Sample code segments of this process are described in Section 4. To understand the complexity of the example in Figure 1, it is useful to examine from where information is collected and how it is manipulated in the process of translating the expression. Information is first collected from the declaration of objects. For example, \texttt{fh} provides information about the number of do loops required and the start and stop values of each loop. Information is also provided by the array expression in that it is used as a template for the loop body. The expression also defines offsets to be made to loop indices through the \texttt{cshift} operator. Furthermore, if the arrays are subscripted then there might be fewer do loops than array dimensions. Finally, information is collected from the context of the array expression; the statement is handled differently if it is within a \texttt{where} statement. In such a case the do loops, communication code, and conditional statements are inserted by the \texttt{where} clause and are not generated at the assignment statement.

The translation sequence is based, as described previously, on the fact that the program is treated as both a single, large expression and, at the same time, a tree based data structure. This is similar to Lisp macros and is described more fully in Section 3. The user overloads the rules, corresponding to nodes in the parse tree, with functions that operate on the sub nodes of the grammar rule. The results are new segments or user defined types that are passed up the expression. Grammar rules that are not overloaded, by default, return the tree associated with the rule. Thus, if no rules are overloaded then the preprocessor passes the program through unmodified.

Functions that operate on parse tree segments can examine or modify other areas of the parse tree expression above the node being currently executed, or the parse trees representing parameters. This feature provides flexibility in making context sensitive decisions on the type of code to generate and the ability to modify remote sections of a program, such as adding declarations or code to initialize data structures. The execution of the parse tree expression results in a new parse tree which is compiled with a standard Fortran compiler.

A detailed description of how the code in Figure 1 is transformed to that in Figure 2 is now given. Before the source statement is executed, the declarations of all the arrays will be executed. For each array, a set of integer variables are declared that define, for each dimension in the array, the first and last element that one thread is responsible for. These are of the form \texttt{ln.xx} and \texttt{hn.xx} for the low and high values, respectively (n indicates the dimension of interest and xx makes the identifier unique). The declaration of a distributed array inserts these declarations at the top of the procedure and, if the array is declared to be static, also inserts code to initialize the variables. If the array is declared to be dynamic then the function that is used to allocate memory, \texttt{allocate}, is responsible for generating code that initializes the variables. Finally, the declaration of an array creates a compile time record containing information about the array. A pointer to this is passed to each use of the declared array.

Before the execution of the source statement, each node in the parse tree is examined to
determine what operation, if any is to be executed. This is done using traditional compiler
techniques and is hidden from the builder of the preprocessor except that the builder must
define the functions and the input and output types. An example of this is given in Section 4.
In the example of Figure 1, the identifier nodes of \( \text{fh}, c, \) and \( \text{qx} \) are identified as distributed
array nodes, and the \(*, +, \text{cshift}, \) and \( = \) operators are identified as array operations.

The execution of the source statement starts at the leaf nodes and works up to the assign-
ment operator. Each of the identifier nodes corresponding to distributed arrays return \( \text{lhs} \)
nodes that are indexed by what will eventually be the index variables of the do loops. Instead
of inserting the identifiers at this time, a marker is put in for each index required and the actual
variables will replace the markers when the assignment operator is evaluated. This is done such
that the \( \text{cshift} \) operator can more easily find the appropriate expression to modify. Note that
the \( \text{cshift} \) operator can operate on general expressions and not just single arrays. The \(* \) and
\( + \) operators do little more than generate a binary expression node consisting of the information
provided.

The bulk of the work is usually done by the assignment operator. If this statement is within
the context of a \( \text{where} \) clause then the assignment statement does nothing more than create
the assignment and pass it up the expression for the \( \text{where} \) clause to generate the appropriate
code. To determine the context, the assignment statement looks up the parse tree for a node
that will execute the \( \text{where} \) transformation code. The context is defined on whether such a
clause is found. If not, the assignment replaces all of the marks within the statement with
index variables, builds the do loops, and adds the communication code.

The number of indices used depends on the declaration of the arrays and the number of
dimensions that are not defined with a scalar value. Thus, the preprocessor also supports F90
triplet notation, although this is not illustrated here. At this point the preprocessor could
check that all of the arrays in the statement have the same shape (the number of non scalar
dimensions is the same) but this is not done. Instead, the assignment statement simply uses
the information provided by the left hand side of the assignment.

The generation of the communication code is fairly straightforward. A template of state-
mements is added after the do loops. The template is parameterized, much like the indices of array
nodes, by such information as array sizes, and local lower and upper bound variables of the
array to be updated. The assignment statement makes a copy of the template, appends it to
the do loops, and replaces all of the marks by the appropriate variable names. This information
is collected from the array in the left hand side of the assignment.

While the described interactions between the various parts of the program are complex, this
is required to translate high level array operations into efficient, parallel, low level code.

3 Implementing The Example With Other Techniques

This section describes several abstraction mechanisms that could be used to implement some-
thing similar to the translation of Figure 1 into Figure 2. The one technique that could imple-
ment the translation described would be to write a general compiler. This would be similar to
the CM-Fortran compiler in all but the code generation section, and would actually be smaller because the portion of the language that the preprocessor recognizes is a subset of CM-Fortran. Unfortunately, such a compiler would be considerably larger than the roughly 3000 lines of code needed to describe the preprocessor using the technology described in this paper. Given only the option to write an entire compiler, we would not have implemented a translator and the user would have had to translate the code manually. There are, however, some techniques in compiler design technology that make this a somewhat more practical problem to solve [GHL+92]. This involves modularizing the entire compiler and the attribute grammar in particular. Given this platform, it would be possible to build the modules such that the user could add the code to implement the desired transformations. However, the user would probably need to understand much of how a compiler works. In comparison, a comparable preprocessor that translates array operations to be executed on sequential machines was built by a student without any formal training in compiler design. Therefore, the preprocessor is probably less complicated to use than the compiler tools.

The difference between a compiler and the preprocessor is that the preprocessor has limited attribution and a fixed model of the base language, along with mechanisms to manipulate and generate code segments. The preprocessor also implicitly handles much of the compilation process. As an example of the limited attribution, generation of indexed array expressions would probably be handled differently in a compiler than that described in Section 2. Instead of passing marked expressions up the tree for later evaluation, information would first be passed down the tree indicating that there was a cshift operator and that the indices needed to be modified accordingly.

Another mechanism for building abstractions is through the use of macros, such as that found in cpp. The similarity to the preprocessor is that both manipulate program segments at compile time. The major difference is that macros have limited ability to either collect or modify information from remote locations in a program. The research area of extensible compilers [SY74] is similar to macro preprocessors in that, for example, the user can define binary operators to be macros. However, the information flow is too complicated for such systems to be able to implement the program translation illustrated in this paper.

Another abstraction technique is that found in object oriented languages. In languages such as C++ it is possible to implement array operations quite easily; a class of distributed arrays is defined and each of the arithmetic operations are overloaded to handle arrays. A problem with this typical approach is that the resulting code is inefficient. Each operator works on an entire array and either generates temporary arrays or writes into other arrays. This can be very inefficient in space usage. This approach also does not use registers efficiently as intermediate values are put back in memory between operations, a costly approach considering the difference between register and memory access times.

A major deficiency with object oriented technologies is that a construct cannot examine information outside of the context of an object. As an example, it would not be possible to implement the where clause because the array assignment operator can not determine the
context in which the assignment is being done.

The next, and most prevalent, form of abstraction is built using subroutines and functions. Semantically, this is similar to that of object oriented technologies and would therefore have roughly the same efficiency characteristics. Syntactically, there is a large difference.

The final abstraction mechanism considered is that of Lisp macros. These are similar to the programmable preprocessor in that the user can specify general compile time operations on program segments. A major difference is that a Lisp macro does not have the ability to make an inquiry into or modify a part of the program that is not one of the parameters to the macro. There are other differences with respect to ease of use, but it is not clear that these substantially change the power to write user directed preprocessing directives. An advantage that lisp macros have over the mechanism described in this paper is that someone that is comfortable writing lisp programs would be able to write the macros, but it is not expected that most users of a preprocessor will be able to write compile time operations. It is expected that someone writing directives for the preprocessor will have a good understanding of programming languages, something that many Fortran programmers do not possess.

4 Overview of Building a Processor

This section summarizes the steps involved in defining a preprocessor and gives some sample code segments of this process. It first describes the form of the parse tree expression and how a user specifies operations that are embedded in it. The typical sequence of operations that an operation performs is then described.

The user's view of the parse tree expression is that each node in the tree corresponds to a rule in an abstract grammar of the base language, Fortran. The fortran abstract grammar has roughly 125 rules and these can be broken into 6 classes. These classes of rules have been picked to closely follow an extended Backus-Naur form (EBNF) notation. This compact form of the grammar is simpler to manipulate than other descriptions, such as BNF. The six classes define how rules can be created; A rule can be built from a set of rules, a list of other rules, an optional rule, one of many rules, a token, or exactly one other rule.

From this representation of the program, the user overloads the rules with procedures that will be called in the order of execution of the tree. This order is from the bottom of the tree up except that lists are executed in a forward order. For example, the lexical order of statements defines the execution order of statements.

Input to each procedure are the sub-trees below the one being executed. Output is a tree of the type defined by the rule. If a rule is not overloaded then it simply returns itself.

The specification of the preprocessor consists of two parts. The first describes which rules to overload and the second describes the operations to be executed on the rules. The specification of how to overload rules is done in one of two ways. The first is to specify the name of a rule and a function that will operate on the rule. With this alone it is possible to implement a preprocessor. But there are special cases that occur so often that there is further support for operations on the expression, function, procedure call, identifier, and operator rules. There is
only enough space in this paper to briefly describe this support. It roughly follows a model of
abstract data types. Objects can be declared and operations can be defined on these objects,
including overloaded Fortran operators.

An example of this is given in Figure 3. This figure is a partial listing of that used in the
example described in this paper. In this figure, keywords are printed in bold font. It starts by
declaring three user defined types, \texttt{refT}, \texttt{arrayT}, and \texttt{SubRange}. The first is an array reference,
the second is an array expression suitable for an array operator, and the third is a range of
indices defined with the triplet operator. The next line defines a coercion that translates Fortran
expressions (type \texttt{BASIC}) into expressions of type \texttt{arrayT}. The identifier following the type, in
this case \texttt{Array}, is the name of a C \texttt{typedef} that represents the object during compilation. The
next three lines define operations on arrays and include index, assignment, and addition. Note
that the index operator is parameterized by an array reference, the array being indexed, and a
list of type \texttt{SubRangeT}.

The example here is similar to that found in object oriented languages such as C++. These
capabilities have been extended in the preprocessor. For example, polymorphic functions can
be defined based on both input and output types. Furthermore, a single function can operate on
variable numbers of parameters. This is used, for example, by a function similar to the Fortran90
\texttt{allocate} function that takes the name of an array to allocate space for, and a variable length
list of integers defining the amount of data to allocate for each dimension in the array. Finally,
there is a mechanism for passing lists of statements as a parameter to a preprocessor function.
This could be used, for example, to build a while loop; the first parameter to the \texttt{while} function
is an expression controlling the loop and the second parameter is the body of the loop.

The execution of a single rule typically consists of collecting information from nodes through-
out the tree, generating Fortran segments based on this information, and placing the segments
in the appropriate places. Due to the nature of the task, these rules are typically written in
C. Each of these tasks is now briefly described and illustrated with the procedure in Figure 4.
This procedure handles array assignment for distributed arrays. Within this figure, types and
functions that are provided by the programmable preprocessor are shown in bold font.

The first task, collecting information, can be done by one of three mechanisms. The first is
through evaluation of the tree. The tree is evaluated in depth first order (lists are evaluated in
forward order) and each node can return a result that is used by the next node up the tree. This

Figure 3: Sample Declaration of Overloading Rules in the Parse Tree Expression
TREE array_asgn( lhs, exp)
    Array lhs, exp;
    
    TREE dotree; /*code containing do loops*/
    TREE body; /*body of do loops*/
    TREE stmt; /*node representing the assignment*/
    int top; /*whether this is in a where clause*/
    int tmres; /*if any marks have been replaced*/
    TREE lhs_tree; /*expression code segment of lhs Array*/
    TREE exp_tree; /*expression code segment of exp Array*/
    
    /*find the rule above this assignment that is a statement rule*/
    stmt = find_up( CurrentNode, cmstRT);
    /*determine if this is within a where clause*/
    top = in_where( stmt);
    
    /*if this is not within a where clause*/
    if( top){
        /*build the assignment statement*/
        lhs_tree = lhs->exp;
        exp_tree = exp->exp
        body = @Stmt{# lhs_tree.exp = exp_tree.exp#};
        /*build the iterator framework*/
        dotree = build_iterator_code( lhs->ref);
        /*replace marks with index vars, etc*/
        do{
            tmres = translate_marks( dotree, lhs->ref, lhs->ranges, body,
            BODY|ARRAY|LCL|INDX|IXL0|IXHi);
        } while(tmres);
        /*replace this node with the do-loop code segment*/
        insert_before( stmt, dotree);
        delete_list_elem( stmt);
        return CurrentNode;
    }
    /*else another node will handle the details*/
    else{
        /*build the assignment code*/
        body = Assign( lhs->exp, exp->exp);
        return body;
    }
    
Figure 4: Code Implementing the Array Assignment Operator
can be a parse tree expression or an internal data structure to be used later on. As an example, the parameters `lhs` and `exp` in Figure 4 are array objects representing the left and right hand sides of the array assignment operator, respectively. The compiler handles the execution order and details of passing the correct parameters.

The second method by which information flows through the parse tree expression is from the declaration of an object to all of the places in which the object is used.

The third method for information flow is through the remote access of one node by another. This is the most general method of moving information and allows any node to collect information from any other node. An example of this are the first two statements in Figure 4. The first statement looks up the parse tree for the enclosing statement grammar rule. This is passed to the second statement that searches up the statement list looking for a `where` statement. This determines whether do loops are to be created or not. There are several functions that can examine parse tree segments but are not described here.

The next task of each function is to generate Fortran code segments based on the information collected. This is very briefly described here. As code segments are represented by trees, one simple method to support code generation is to define a function for each grammar rule or node type in the language. A code segment can then be created using nested function calls. An example of this is the generation of an `Assign` rule in the second to last statement. However, this tends to be very cumbersome in all but trivial cases, so an additional mechanism has been provided. This allows the module writer to write Fortran segments such as expressions and statements. These segments can be parameterized with other segments. A simple example of this is the creation of the body of the do loop in Figure 4. The assignment to body builds a code segment that is an assignment statement of the indexed left and right hand sides. The two parameters to this include `lhs-tree` and `exp-tree`. Both of these are code segments of type expression, as defined by the appended `.exp`.

The final task is to place the generated code segments in the correct places in the expression tree. Typically, the code segment returned by a function replaces the node that generated the segment. The base compiler handles such cases. An example of this is illustrated in the last few lines of Figure 4. If the assignment is within a `where` clause then this function should just return an assignment rule. This is done by first building the assignment and then assigning it to `body` and returning this value.

Occasionally, the location of where to put the translated code is elsewhere. In Figure 4, this is illustrated in the placement of `dotee`, the generated code including loops and the assignment statement. In this case the generated code, a list of statements, must replace the statement that contains the array assignment this function is translating. To do this, the function `insert_before` is used to first insert the new code before the statement containing this assignment, and then the function `delete_list_elem` is used to delete this statement. The underlying compiler deals with the fact that the section of parse tree expression that is currently being executed is deleted. There are several other routines for finding tree nodes of certain types and for inserting and deleting nodes into parse tree expressions.
5 Conclusion

This paper describes a programmable preprocessor for quickly building high level abstractions that can be translated into efficient code. This is done by breaking the usual compilation sequence into two phases. The first handles most of the compilation details. The second, and new phase, treats the program as a single, large expression, that when executed returns a transformed version of the source program.

Using this technology, several preprocessors have been built. The largest, around 3000 lines of code, supports user defined mapping functions, Fortran90 dynamic arrays and array operations, and CM-Fortran cshift operators. Another preprocessor was built to check parameter types of procedures and functions across multiple files. A preprocessor was also developed by an undergraduate student to implement array operations using triplet notation. The preprocessor generates code to be run on a scalar machine and is roughly 800 lines long.

References
