JOURNAL OF COMPUTATIONAL PHYSICS **138,** 981–990 (1997)
ARTICLE NO. CP975813

NOTE

The LASY Preprocessor and Its Application to General Multidimensional Codes

1. INTRODUCTION

In this paper a new preprocessor using the *loop annotation syntax* is described. One of the possible applications of this syntax is writing a general simulation code where the number of grid dimensions is a parameter of the preprocessor. Programming language constructs, which would be normally repeated for each dimension separately, are annotated as loops in the syntax, and these loops can be expanded by the preprocessor. The preprocessor overcomes some of the inherent limitations of most computer languages, such as the fixed number of indices for arrays. In general the typical expressions used in multidimensional simulation codes can be written in a way that is more concise and less prone to error. The preprocessor is implemented in the Perl language which is available for almost all operating systems.

The loop annotation syntax (LASY) was developed as part of the versatile advection code (VAC) [1, 2] software package which can solve conservation laws, e.g. of hydrodynamics and magneto-hydrodynamics on one, two, and three-dimensional grids. The usual practice is to write a simple 1D code first, then to modify it for 2D simulations, and finally, years later, to rewrite the whole code for 3D. In the VAC project a different route was taken, namely a single general software was designed from the beginning, which can do simulations in any number of dimensions. (The acronym LASY should be pronounced with a z, since laziness if "*the first great virtue of a programmer*" [3]).

In object oriented languages, such as C++, it would be possible to define objects which are arrays with N_{dim} dimensions (where N_{dim} is a parameter) and then to define operations acting on these objects. The natural choice for the source language of VAC was, however, Fortran because extensive libraries and efficient compilers are available, and because the potential users of the code are familiar with Fortran but not with C++.

To circumvent the limitation that the number of array dimensions has to be known by the Fortran compiler, the LASY preprocessor translates the dimension independent source program to a standard Fortran code with a given number of dimensions N_{dim}, which is an adjustable parameter for the preprocessor. The preprocessor does not try to understand the semantics of the code, it simply generates the correct syntax by repetition and substitution. For example, the three indices of an array can be regarded as three variants of the same general index string separated by commas. Similarly the sum of these indices can be interpreted as the three substitutes of the general index separated by plus signs. Unlike the object-oriented approach, where one has to define a new operation or generalize an existing one for all the possible cases that involve dimension independent objects, in the preprocessor approach one needs to recognize the loops in the syntax and annotate them appropriately.

During the course of the software development it was found that the preprocessor can be efficiently used for other loop-like expressions as well, such as operations on the minimum and maximum of certain values. Thanks to the very general rules of LASY this simply required the introduction of a few new preprocessor variables with two substitutes, e.g. the strings `min` and `max`. The preprocessor can probably have numerous different applications, some of which may require slight modifications or extensions of the basic rules, but the concept of syntax loops seems to be very powerful.

It is also interesting to note that the translated code is usually quite efficient. One can, for example, use N_{dim} scalars `n1`, `n2` ..., instead of an array with N_{dim} elements `n(ndim)`, which would be the general solution in Fortran. While it takes some time to get used to the LASY notation, it becomes quite natural with experience. The fact that the rather complex VAC software, which is approximately 10,000 lines long, could be designed, implemented, and tested in about one year, proves that the preprocessor is practical and reliable. The preprocessor has been implemented in the Perl language [3] which is available for almost all computers and operating systems. The source code of the LASY preprocessor, which is about 300 lines, and a detailed description of the actual implementation are freely available from the author.

In Section 2 the general loop annotation syntax is defined for simple syntax loops, while Section 3 presents rules and examples for more complex cases. In Section 4 some practical considerations of the implementation are discussed. Conclusions based on my experience with LASY are given in Section 5.

The examples presented in this paper are based on the source code of the versatile advection code. The VAC source uses more features and variables of LASY than I describe here, and the examples shown in the paper are not meant to be the most efficient or elegant way of solving the problems, they simply demonstrate the use of LASY.

2. SIMPLE SYNTAX LOOPS

The LASY preprocessor extends the syntax of the original computer language, which I will refer to as the base language. The extensions should not interfere with the syntax of the base language, therefore the preprocessor directives are either disguised as comments in the base language, or the extensions are distinguished by some special characters, e.g. "#," that do not occur in the base language. LASY uses the latter approach, and the special characters chosen do not occur in Fortran 90 (or Fortran 77) statements, except for quoted text or comments. This choice is not wired into the preprocessor at all. The special characters can easily be replaced by others, and even by longer strings although the latter may require some minor modifications. In the following I use the special characters that were chosen for the Fortran base language.

The most important special character in LASY is ^ which precedes the preprocessor variables. Such a variable represents a number of substitute strings, thus it will be referred to as a *pattern*. For example the

$$\texttt{^D} \rightarrow \underbrace{1, 2, \ldots}_{N_{\text{dim}}} \tag{1}$$

rule simply means that the `^D` pattern is replaced by either 1, or 1 and 2, or 1, 2, and 3, depending on the value of N_{dim}. Another rule may have a fixed number of substitutes, e.g.,

$$\texttt{^LIM} \rightarrow \texttt{min,max} \tag{2}$$

In general a pattern consists of the special pattern character ^ followed by one capital letter, followed by zero or more characters which can be capital letters or the characters & and %. More formally, using the notation of regular expressions:

$$\textit{pattern} ::= \texttt{^[A-Z][A-Z\&\%]}* \tag{3}$$

This particular choice of the character sets (denoted by [...]) is not an essential feature of LASY, but it was convenient for our applications. Capital letters are convenient since the Fortran code can be written in lower case letters, and the base language and the language extensions are then well separated visually. The use of the % and & characters will be explained later.

Patterns in general are defined by their substitution rules

$$\textit{pattern} \rightarrow \texttt{sub1, sub2, ..., subN} \tag{4}$$

where `sub1, sub2, ..., subN` are N arbitrary strings, and N is a nonnegative integer. These substitution rules are defined in the initialization part of the preprocessor, and rules can be easily modified or added according to the needs of the application. Choosing good pattern names and substitution strings is a responsibility of the programmer. In this paper the rules used for the VAC source code are shown as particular examples.

The patterns behave like the index variables of the syntax loops. The most complete form of a syntax loop consists of a special opening character "{" followed by program text `txt` in the base language mixed with patterns, then a special separator character "|" which precedes the separator string `sep`, and finally a special closing character "}":

$$\textit{loop} ::= \{\texttt{txt}\,\textit{pattern}\,\texttt{txt} \mid \texttt{sep}\} \tag{5}$$

A loop can extend over several lines, and the opening and closing curly braces have to match. The substitution rule for the loop is

$$\textit{loop} \rightarrow \texttt{txt sub1 txt sep txt sub2 txt sep} \cdots \texttt{txt subN txt} \tag{6}$$

To put it in words, the part between "{" and "|" is repeated as many times as the pattern has substitutes. In each repetition the corresponding substitute of the pattern is used, and the repetitions are separated by the separator string enclosed by the "|" and "}" characters. The following example shows three syntax loops with Fortran base language and their expanded forms using the pattern definitions (1) and (8) for the case $N_{\text{dim}} = 2$:

$$\begin{array}{ll} \texttt{\{do ix\^{}D = 1, 10|; \}} \rightarrow & \texttt{do ix1 = 1, 10; do ix2 = 1, 10} \\ \quad \texttt{w(\{ix\^{}D|, \}) = 0} \rightarrow & \quad \texttt{w(ix1, ix2) = 0} \\ \texttt{\{enddo\^{}D\&|; \}} \rightarrow & \texttt{enddo; enddo} \end{array} \tag{7}$$

There is a new pattern `^D&` in the last line with the substitution rule

$$\texttt{\^{}D\&} \rightarrow \underbrace{\text{'', '', ..., ''}}_{N_{\text{dim}}} \tag{8}$$

i.e., it is substituted by N_{dim} empty strings, denoted by ''. The purpose of introducing this pattern is to define the number of repetitions, in this case the number of `enddo`'s, separated by semicolons. The use of the & character in the pattern name `^D&` is a matter of taste, it could have just as well been called `^DEMPTY` or `^DTIMES`.

Although the dimension independent notation in (7) is already shorter and more general than the expanded form for two dimensions, it can be simplified further. In LASY there are two ways of defining the separator string without using the special character "|" in the loop. If some common separator character s is found at the very end of the loop, it is interpreted as a single character separator string. Otherwise the comma is used as default separator, which is the most commonly needed separator in Fortran,

$$\begin{array}{l} \{\texttt{txt}\,\textit{pattern}\,\texttt{txt}\,s\} \rightarrow \texttt{txt subl txt}\,s\,\texttt{txt sub2 txt}\,s \cdots \texttt{txt subN txt} \\ \{\texttt{txt}\,\textit{pattern}\,\texttt{txt}_s\} \rightarrow \texttt{txt subl txt, txt sub2 txt,} \cdots \texttt{txt subN txt} \end{array} \tag{9}$$

where $s ::= [\sqcup + - * /, :; \setminus]$ for the particular version of LASY used for the VAS source code, and $\sqcup$ denotes the space character. The subscript in $\texttt{txt}_s$ simply means that the last character cannot be one of the common separator characters. The $\setminus$ separator character is replaced by a new line in the output. Note that the separator can still be defined as an empty string with the full form (5) of the loop, i.e. {`txt` *pattern* `txt`|}.

A further and more important simplification of the notation is that the preprocessor can expand patterns without the enclosing special characters “{” and “}”. A pattern in itself implies a loop which is bounded by the typical delimiter characters of the base language. For Fortran the following characters were chosen: space, comma, semicolon, and enclosing parentheses found at the same nesting level of parentheses, and the beginning or the end of the line. By a formal definition, an implied syntax loop is

$$\begin{aligned} \mathit{implied\_loop} ::= {} & l\,\mathtt{txt}_l\,\mathit{pattern}\,\mathtt{txt}_r\,r \\ & l\,\mathtt{txt}_l\,\mathit{pattern}\,\mathtt{txt}_r\,\mathtt{sep}_r\,r \end{aligned} \tag{10}$$

where $l ::= [\sqcup, ;\, (]$ and $r ::= [\sqcup, ;\,)]$, and the l and r indices mean that `txt` or `sep` cannot contain any left and right limiter characters unless they are enclosed in parentheses. An implied loop cannot extend over more than one line. Implied loops are expanded the same way as loops delimited by curly braces (see (6)), except that the bounding characters l and r, which belong to the base language, are retained. It was also found convenient (see (23)) to force the separator to be a semicolon when the right delimiter is a semicolon, i.e.

$$l\,\mathtt{txt}\,\mathit{pattern}\,\mathtt{txt;} \rightarrow l\,\mathtt{txt\,sub1\,txt;\,txt\,sub2\,txt;}\cdots\mathtt{txt\,subN\,txt;} \tag{11}$$

As an example here are some variable declarations and a Fortran do loop similar to (7):

```
integer, parameter :: nx^D = 10 → integer, parameter ::
                                  nx1 = 10, nx2 = 10
integer :: ix^D                 → integer :: ix1, ix2
real :: w(nx^D)                 → real :: w(nx1, nx2)
{do ix^D = 1, nx^D\}            → do ix1 = 1, nx1
w(ix^D) = 0.                      do ix2 = 1, nx2
enddo^D&\                       →   w(ix1, ix2) = 0.
                                → enddo
                                  enddo                  (12)
```

At a level of abstraction higher than those of patterns and syntax loops, one may regard `nx^D` and `ix^D` as the generalized multi-D array dimension and index, respectively.

LASY can perform the most basic preprocessor tasks as well. A constant variable can be defined as a pattern with a single substitute string, while the conditional inclusion of program text is obtained by a syntax loop which includes the program text and a pattern that has 0 or 1 nulstring substitutes depending on the condition. Two simple examples are

$$\texttt{\^{}ND} \rightarrow \quad N_{\text{dim}} \tag{13}$$

$$\texttt{\^{}IFONED} \rightarrow \quad \underbrace{\text{`'}}_{0 \text{ or } 1} \tag{14}$$

where the number of substitute empty strings for `^IFONED` is 1 for $N_{\text{dim}} = 1$ and 0 otherwise. See (25) and (22) for examples of usage.

3. MULTIPLE SYNTAX LOOPS

The possibility of having more than a single kind of patterns triggers a number of questions. What happens if there are more, possibly different, patterns in a loop? What about nesting syntax loops? What happens if a substitute string contains further patterns? The answers to these questions are given in the following LASY rules:

- The number of repetitions is determined by the *first* pattern in the (implied) syntax loop, and only patterns with the *same initial letter* are substituted.
- In the case of nesting, the outermost loop is expanded first as if it was a simple loop.
- The expansion continues until no patterns are left.

These rules were formulated to accommodate the applications but one may find some further justification for them. Since LASY is syntax oriented, it is reasonable that the rules do not refer to the meaning of the patterns (one could have introduced some precedence of patterns, for example). Both the first and the second rules imply a natural left-to-right translation order, and this translation is repeated, if necessary, according to the third rule. The following implied double loop uses the pattern definitions (1), (2), and it expands according to the above rules:

$$\begin{aligned}\texttt{w(ix\^{}LIM\^{}D:)} &\rightarrow \texttt{w(ixmin\^{}D : ixmax\^{}D)}\\ &\rightarrow \texttt{w(ixmin1 : ixmax1, ixmin2 : ixmax2)}\end{aligned} \tag{15}$$

An even more compact way of writing array *segments* is by introducing the `^S` pattern

$$\texttt{\^{}S} \rightarrow \underbrace{\texttt{\^{}LIM1:, \^{}LIM2:, ...}}_{N_{\text{dim}}} \tag{16}$$

and thus the third rule implies

$$\begin{aligned}\texttt{w(ix\^{}S)} &\rightarrow \texttt{w(ix\^{}LIM1 :, ix\^{}LIM2:)}\\ &\rightarrow \texttt{w(ixmin1 : ixmax1, ixmin2 : ixmax2)}\end{aligned} \tag{17}$$

In loops where one of the repetitions should be very different from the others, a *selector* starting with the special characters ^% is used:

$$\textit{selector} ::= \texttt{\^{}\%}I \tag{18}$$

where $I = 1, \ldots, N_{\max}$ is a positive integer, and $N_{\max}$ is the maximum number of substitutes for all defined patterns. Loops that contain a selector will expand the text (with patterns) that precedes the selector in the I^{th} repetition and will expand the text (with patterns) that follows the separator in all the other cases. Formally, although not in its most general form, the substitution rule is

$$\begin{aligned}&\texttt{txt}_I\,\textit{pattern}\,\texttt{txt}_I\texttt{\^{}\%}I\,\texttt{txt}_O\\ &\quad\to \underbrace{\texttt{txt}_O, \texttt{txt}_O, \ldots,}_{I-1}\ \texttt{txt}_I\,\texttt{subI}\,\texttt{txt}_I, \underbrace{\texttt{txt}_O, \texttt{txt}_O, \ldots}_{N-1}\end{aligned} \tag{19}$$

In general the first *pattern* may occur after the *selector*, or there can be patterns (with identical initial letters) both before and after the selector, then $\texttt{txt}_O$ will be mixed with substitutes `sub1, ..., subN` with the exception of `subI`. The loop may be defined by special characters (5) or implied by delimiter characters of the base language (10), and the separator string can be defined as in (6), (9), or (11).

In most cases the selectors are not used directly, but they occur rather as substitutes of some other pattern, e.g.

$$\texttt{\^{}D\%} \to \underbrace{\texttt{\^{}\%1, \^{}\%2, }\ldots}_{N_{\text{dim}}} \tag{20}$$

As an application I show a short piece of Fortran 90 code that sets certain elements of the `w` array to zero. The selected array elements are located at the two grid boundaries orthogonal to the `idim` direction. The main syntax loop is for the `^D` pattern, next the loops implied by the `^S` patterns are expanded in two steps

```
select case(idim)               → select case(idim)
{case(^D)                       → case(1)

  w(ixmin^D^D%ix^S) = 0.             w(ixmin1^%1ix^S) = 0. →

  w(ixmax^D^D%ix^S) = 0.\}           w(ixmax1^%1ix^S) = 0. →

end select                        case(2)

                                     w(ixmin2^%2ix^S) = 0. →

                                     w(ixmax2^%2ix^S) = 0. →

                                → end select
```

(21)

```
   select case(idim)           select case(idim)
   case(1)                     case(1)
   w(ixmin1,ix^LIM2:)=0. → w(ixmin1,ixmin2:ixmax2)=0.
   w(ixmax1,ix^LIM2:)=0. → w(ixmax1,ixmin2:ixmax2)=0.
   case(2)                     case(2)
   w(ix^LIM1:,ixmin2)=0. → w(ixmin1:ixmax1,ixmin2)=0.
   w(ix^LIM1:,ixmax2)=0. → w(ixmin1:ixmax1,ixmax2)=0.
   end select                  end select
```

To end this session I present another idiomatic LASY expression, the shift of array segments in the `idim` direction. Although this could be achieved with the above case construct as well, introducing a global $N_{\text{dim}} \times N_{\text{dim}}$ Kronecker delta matrix

```
integer :: kr(^ND, ^ND)    → integer :: kr(2,2)
kr = 0; {kr(^D, ^D) = 1;} → kr = 0; kr(1, 1) = 1;
                                        kr(2, 2) = 1  (22)
```

somewhere in the initialization part of the program can be used in a single line expression for the shift:

```
jx^LIM^D = ix^LIM^D + kr(^D, idim); →
                            jxmin1 = ixmin1 + kr(1, idim);
                            jxmin2 = ixmin2 + kr(2, idim);
                            jxmax1 = ixmax1 + kr(1, idim);
                            jxmax2 = ixmax2 + kr(2, idim); (23)
```

The intermediate step of the expansion of the `^LIM` implied loop is not shown, and the rather long expanded line is broken into shorter pieces for clarity.

4. IMPLEMENTATION OF THE LASY PREPROCESSOR

The preprocessor has to take care of quoted text. In my implementation of the LASY preprocessor the first bit of all the characters of the quoted text is set to 1 before any processing is done, and just before the output is printed the first bit is set back to 0. In this way the characters in the quoted text have ASCII codes above 127 during the preprocessing, and there can be no confusion with the special characters of LASY.

Another advantage of hiding the quoted text is related to the formatting of the output. The length of a line can expand significantly during the preprocessing, so it may need to be broken into shorter pieces, preferably at well chosen breaking points. The best breaking points are guessed by looking for characters like [+ − */,] near the intended location of the line break. The original quoted text may easily contain these characters, but a line may not be broken in the middle of a quotation. The quotation-hiding mechanism outlined above prevents this happening.

Error messages and warnings of the base language compiler always refer to line numbers of the preprocessed code which is usually different from the corresponding line number in the original code. Thus the expanded code should be easy to read. Therefore the original indentation is retained and the preprocessor attempts to break the lines into more or less logical parts. Good formatting of the output helps the user to understand the preprocessed code.

The preprocessor also provides the facility of including other LASY files. A line containing

```
INCLUDE : file
```
(24)

will be replaced by the contents of `file`. Included files may contain references to further included files themselves. The inclusion of the file may be made conditional by enclosing it with a syntax loop

```
{^IFONED INCLUDE : file}
```
(25)

Note that the syntax of `INCLUDE:` is intentionally different from the include statement of the Fortran base language.

The definition of patterns is fairly simple in the Perl source code of the preprocessor. The `patdef` Perl subroutine is called with the pattern name, the number of substitutes, and a list of substitutes with an arbitrary number of elements. The extra elements are simply ignored, while missing elements are taken to be empty strings. Here are the subroutine calls that correspond to the pattern definitions (1, 2, 8, 13, 14, 16, and 20)

```
$ndim = 2;
&patdef('D'      , $ndim      , 1        , 2        , 3        );
&patdef('LIM'    , 2          , 'min'    , 'max'               );
&patdef('D&'     , $ndim                                       );
&patdef('ND'     , 1          , $ndim                          );
&patdef('IFONED' , $ndim == 1,                                 );
&patdef('S'      , $ndim      , '^LIM1:' , '^LIM2:' , '^LIM3:');
&patdef('D%'     , $ndim      , '^%1'    , '^%2'    , '^%3'    );
```
(26)

where the `$ndim` == 1 expression equals 1 if `$ndim` is 1, and 0 otherwise.

5. CONCLUSIONS

Writing a LASY code is not as complicated as it may seem for some of the examples. In the overwhelming majority of cases the simplest implied loops are used with the most common patterns like `^D` or `^S`. More complicated expressions can be easily checked by running the preprocessor interactively; i.e., the LASY code is typed in via standard input and the translated code appears on the standard output. After a loop is found to work properly it soon becomes an idiomatic expression which can be remembered without thinking about the details of the expansion. In writing multidimensional codes a typical source of bugs results from copying an expression for a certain dimension or index several times and forgetting to do all the necessary changes afterwards. In LASY this cannot happen since the repetitions and modifications are done by the preprocessor automatically and free of typing errors.

Since LASY is an extension of the base language one may add code in the base language without difficulty; e.g., subroutines for a particular number of dimensions can be written without knowledge of the loop annotation syntax. This is an important feature for users of the code who do not need to learn LASY.

For the programmer LASY offers a compact, efficient, and general notation, which can be learned easily and extended or modified according to the arising needs; therefore it truly serves the virtue of *laziness*.

ACKNOWLEDGMENTS

The software development of LASY and VAC was done as part of the project of massive parallel processing applied to magnetohydrodynamics, which is funded by the Dutch Science Foundation (NWO). The author also received partial support from the Hungarian Science Foundation, Grant F 017313.

REFERENCES

1. G. Tóth, A general code for modeling MHD flows on parallel computers: Versatile advection code, *Astrophys. Lett. & Comm.* **34,** 245 (1996).
2. G. Tóth, Versatile advection code, in *Lecture Notes in Computer Science, Proceedings, HPCN 97', Vienna, Austria, April 28–30, 1997,* (Springer-Verlag, New York/Berlin, 1997), Vol. 1225, 253.
3. L. Wall and R. L. Schwartz, *Programming Perl* (O'Reily, Sebastopol, USA, 1992), p. 426.

Received July 19, 1996; revised February 12, 1997

Gábor Tóth
Sterrenkundig Instituut
Utrecht, The Netherlands
E-mail: toth@fys.ruu.nl

Present address: Dept. of Atomic Physics, Eötuös̈ University, Budapest, Hungary. E-mail: gtoth@hercules.elte.hu.http://tys.ruu.nl/~toth.