Translating non-standard extensions to standard Pascal

by VISWANATHAN SANTHANAM
Wichita State University
Wichita, Kansas

INTRODUCTION

Extensions to a programming language are often introduced by individual implementors for a variety of reasons. In some cases, a new construct may allow the programmer to take advantage of a unique hardware or software feature in the specific system. In other cases, the application for which the language is frequently employed may warrant simpler and more efficient constructs. Whatever the motive, extensions, in general, severely limit portability of programs. Often it is necessary to manually translate non-standard features before a program can be successfully adapted to a new environment. If we must live with such extensions at all, ideally we would like to be able to translate all extensions automatically to make the adaptation expedient and reliable.

Some extensions clearly do not have direct translations in all target environments. Others lend themselves to system-independent translations, e.g., to the base (standard) language itself. Typically such translations do not have the same elegance and efficiency as the original construct, but they enhance the portability of programs. A system which accepts descriptions of such extensions and their translations can automatically process non-standard programs, yield a standard translation and provide a precise definition of the extension to a naive reader of the program. In this paper, we outline such a system that we call “extension processor.”

The extension processor is somewhat similar in function to a macro processor in that both systems attempt to preprocess the source text of a program and produce an output suitable for the compiler. There are a number of macro processors designed for use with Pascal-like languages (see References 3 and 4 for examples) and some of these are powerful enough to handle minor extensions such as the use of constant expressions where only simple constants are permitted in the base language. But these systems are not capable of describing more complex forms of translations that may involve source code movement and/or augmentations far from the point where the extension is used. The proposed extension processor can effectively describe moderate forms of global and local transformations that are sufficient for translating many popular extensions to Pascal.

THE EXTENSION PROCESSOR

Extensions are described by the programmer with the help of two new statements—GRAMMAR and TRANSLATE. The GRAMMAR statement describes the syntax of the extensions, while the TRANSLATE is used to specify the standard language translations for them. Each of these statements can be included in any Pascal block, preceding all other declarations such as LABEL, CONST, etc. Like all other declarations, GRAMMAR and TRANSLATE are also subject to the scope rules of Pascal. This means an extension can be “global” if defined at the program level, or “local” if defined in a lower level block. It is also possible to translate the same extension in different ways in different blocks. The complete syntax of these statements is outlined in the appendix. The overall structures and the semantics are discussed in this section.

The main structure of the GRAMMAR statement is depicted in the syntax diagram below.

Here (non-term) is the syntactic identifier of the extension feature; (substitution) is merely a linear representation of a syntax diagram for that extension. The (substitution) itself is a sequence of tokens consisting of key words (both standard and user defined), operators and other non-terminals, defined elsewhere in a GRAMMAR statement. Not all non-terminals need to be explicitly defined; “standard” definitions could be included within the extension processor, just as standard type identifiers, such as INTEGER, BOOLEAN, etc., are defined in Pascal.

In addition to the terminals and non-terminals, a substitution may also include labels to identify syntactic entities in the definition. These labels are useful in describing the translation later in a TRANSLATE statement.

There are two special operators provided in the GRAMMAR statement: the “follow-on” operator → which is op-
tionally employed to clarify the physical juxtaposition of two successive syntactic entities; and the "alternates" operator \(\rightarrow\) which indicates a branch-off point in the syntax diagram.

For example, the syntax diagram

\[
\text{THIS} \rightarrow \text{NOT} \rightarrow A \rightarrow \text{GOOD} \rightarrow \text{EXAMPLE} \rightarrow \text{BAD}
\]

can be expressed in a GRAMMAR statement like this:

\[
\text{GRAMMAR}
\]

\[
.\text{example1} \rightarrow \text{THIS} =: (. \text{NOT} ), .(
\]

\[
\rightarrow A =: (. \text{GOOD} ), (. \text{BAD} ), \rightarrow \text{EXAMPLE};
\]

Recursive definitions are written in a manner similar to the BNF notation. For example, this syntax diagram

\[
\text{THIS} \rightarrow \text{IT} \rightarrow \text{IT}
\]

is translated to a GRAMMAR statement by defining a separate non-terminal for the recursive component "THIS":

\[
\text{GRAMMAR}
\]

\[
.\text{example2} =: \text{.t.} =: \text{.t.} \rightarrow \text{"see definition below"}
\]

\[
\rightarrow \text{.AND. \&.t.} =: \text{\"the escape option\"} \rightarrow \text{\"the recursion option\"} \rightarrow \text{IS IT};
\]

The TRANSLATE statement is more complex as it must describe the more difficult task of translating the extension to the standard language. Many times there is a need to add variables or type identifiers to the local block or the program block. Sometimes it may also be necessary to add executable statements to the body of a procedure. The TRANSLATE statement syntax provides for these specifications in Pascal-like constructs. The overall syntax diagram for the statement has the form:

\[
\text{<translate>}
\]

The non-terminal referenced above must be defined in a GRAMMAR statement, either explicitly or implicitly as a standard non-terminal. The (translation) consists of three components—"global includes," "local includes" and "substitution." The global and local "includes" are intended to specify new labels, identifiers, procedures, functions or statements that may be needed at the program level and in the current block respectively. Thus each of these "includes" is divided into eight parts: LABEL, CONST, TYPE, VAR, PROCEDURE, FUNCTION, TOPOFSTMT and BOTOFSMT.

The syntax of each part, so far as the extension processor is concerned, is the same. However, the phrases constructed by the extension processor must be acceptable to the Pascal compiler. The actions of the extension processor itself are restricted to adding the phrases specified in each part to the appropriate parts of the program or local block.

The substitution part is identified by the key word TRANS and it describes the in-place substitution for the recognized extension. The description takes the form of a sequence of terminals (operators, key words etc.), non-terminal labels (defined in the GRAMMAR for the extension), and identifiers and labels defined in the "includes" (called "sysnames" and "syslabels"). Special built-in functions called "sysfunctions" are also provided for generating tokens that do not appear explicitly within the extension text, but are "computable" by the extension processor. For example, &EXPTYPE($E) generates the type of the expression $E.

Furthermore, three powerful "syscontrol" statements are also provided for the control of the number and sequence of tokens generated; these are &FOR (also, &FOREACH), &IF and &CASE. These syscontrols are similar in function to the conditional compilation features of many macro processors.

The sysfunctions and syscontrols together form the heart of the translation language used by the extension processor. By adding more powerful features into these two categories, it is possible to widen the range of extensions that can be translated. In the following section, we illustrate the use of some of the more basic features of this work.

EXAMPLES

A set of five examples is presented in this section to illustrate features of the extension processor.

Example 1: the extended CASE statement

Many implementations of Pascal, such as Reference 1 for example, have extended the CASE statement to accept an "OTHERWISE" phrase which specifies an action for cases not covered in the body of the main CASE. This example illustrates how such a CASE statement can be translated to the standard language.

Even though it is not necessary, we will assume that the extended CASE statement begins with a different word, say ECASE. This change makes it easier to describe the extension as a new statement and the parser within the extension processor can recognize it more easily. With this modification the typical extended CASE construct can be described in a syntax diagram as follows.
Translating Non-'standard Extensions to Standard Pascal

This syntax is described in a GRAMMAR statement as follows.

**EXAMPLE:**

```
<loopstatement>
  LOOP -> <statement list>
  | EXIT IF <expression> ;
  | <statement list>
  | END
```

The &FOR $CASES is really not needed as there is only one $CONL in the GRAMMAR for ECASE. It is included here for clarity that the inner &FOREACH is confined to the non-terminal $CASES.

**Example 2: the LOOP statement**

The NBS Pascal compiler introduces the notion of a LOOP statement to allow efficient construction certain classes of loops without the use of the GOTO statement. (The NBS compiler does not itself implement the GOTO.) The syntax of this extension is described by

```
<loopstatement>
  LOOP -> <statement list>
  | EXIT IF <expression> ;
  | <statement list>
  | END
```

The translation of the LOOP statement is straightforward if the use of GOTO statements is permitted.

```
TRANSLATE .loopstatement. TO
  LOCAL LABEL &2; "new label needed for GOTO"
  TRANS
    BEGIN (* loop *)
      &X := $EXP; (* compute and save case expression *)
      IF &X IN [ "now, list all constants"
        &FOR $CASES &DO "begin looking at $CASES"
          (. &FOREACH $CONL &DO
            .($CONL "list the constants"
              &IFMORE.( , ). "comma if more to go"
            ). ) "ending bracket for the set"
          THEN (* if one of the constants *)
            CASE $EXP OF "do a standard CASE"
              $CASES "list the cases as is"
          END (* CASE *)
      ELSE $OTHSTMT "the otherwise part"
      END (* extended CASE *)
    TREND; "end translation"
```

This translation may be unacceptable if GOTOs are not implemented in the target system either. A translation without
the GOTO can also be devised, though somewhat belaboredly.

TRANSLATE .loopstatement. TO "the goto-less version"
LOCAL "define a local procedure for $SL 1"
PROCEDURE &SL1;
BEGIN
$SL1
END;
TRANS "now, use &SL1 for $SL1 and unfold the loop"
BEGIN
&SL1; (* leading execution of $SL1 *)
WHILE NOT ( $EXP ) DO
BEGIN
$SL2;
&SL1 (* top of the loop here again *)
END
END (* LOOP *)
TREND;

Example 3: conditional expressions

Algol’s conditional expressions can save coding time and improve execution speed of programs. Pascal does not include this feature in the spirit of keeping the language simple. We can readily introduce this feature for simple data types as an extension.

GRAMMAR .expression. = : (. .conditionalexp. ).; "add to expressions" :;
 .conditionalexp. = : CIF "choose a different prefix for ease"
 $BOOL .: expression.
 THEN ( $EXP1 .: expression. )
 ELSE ( $EXP2 .: expression. ) ;

We translate the entire conditional expression into a function call, which then returns one of the two values, $EXP1 or $EXP2, depending on $BOOL.

TRANSLATE .conditionalexp. TO LOCAL
 FUNCTION &F : &TYPE.( $EXP ) ;
 BEGIN
 IF $BOOL THEN &F := $EXP1
 ELSE &F := $EXP2
 END;
 TRANS "the in-place substitution is simple"
 &F
 TREND;

Example 4: macro statements

The extension processor described in this paper does not necessarily replace the functions of a macro processor. A simple text substitution type macro processor can express certain preprocessing requirements better than our extension processor can. The most logical place for a macro processor is between the extension processor and the compiler; that is, the output from the extension processor can be the input to the macro processor whose output in turn will be the compiler’s input. This means that any non-standard construct that is to be passed on to the macro processor must be left alone by the extension processor. In particular, the MACRO statement itself must be acceptable to the extension processor. This can be achieved by stating the syntax of the MACRO statement in a GRAMMAR statement at the program level. An example of such a declaration follows.

GRAMMAR .declarations. = : = : (. .macrodcl. ).; "add to declarations" ;
 .macrodcl. = : MACRO .macdefs. ;
 .macdefs. = : $MACNAME .: identifier .: .0.,
 "case of no parameters"
 (. .idlist. ).; "with parms"
 = # .: terminals. # ; "prototype"
 - : : .0.,
 "end of macros"
 (. .macdefs. ).; "more to go"
 :: "end of MACRO syntax".

Additional definitions may be needed if the macro parameters are represented by special identifiers.

Example 5: the BUILTIN attribute

In PL/1 it is possible to return to the global level meaning of an identifier, even though a surrounding block may have redefined it differently. The BUILTIN attribute of PL/1, which does this, is somewhat restrictive, but the concept is quite useful. A similar facility could be made available in Pascal if we define a key word "BUILTIN" and allow declarations such as

```pascal
TYPE BOOLEAN = BUILTIN;
```

A similar use with the CONST declaration may also be permitted. The extension processor can be called in to convert this non-standard feature into standard Pascal by defining a new identifier at the program level, equating it to the standard identifier there and substituting the new identifier in place of BUILTIN. The following GRAMMAR and TRANSLATE statements achieve this.

GRAMMAR
 .constphrase. = : = : (. .builtinphrase. ).; ;
 .typephrase. = : (. .builtinphrase. ).; ;
 .builtinphrase. = : $X .: identifier .: = BUILTIN ;
TRANSLATE .builtinphrase. TO GLOBAL
 TYPE &GLOBALTYPE = $X;
 TRANS
 $X = &GLOBALTYPE
 TREND;
The foregoing examples only illustrate the simpler applications of the extension processor. A number of features, such as the &CASE and &IF syscontrols, were not needed in these situations. Both &CASE and &IF are useful when it is necessary to generate different translations for a syntactic entity depending on its alternates in the grammar.

CONCLUSION

We have outlined an "extension processor" that will allow a Pascal user to automatically translate many non-standard extensions into standard Pascal. In addition to making the adaptation of program using extensions easy and reliable, it also provides a way to define one's own extensions, more powerful than possible with a macro processor.

The question of "what is standard Pascal?" is not at issue in this paper. It is only necessary to have a working definition of a "base" language whose features are generally accepted as standard. We have assumed no single definition as the standard language; but all the features used in the illustrative translations are a subset of the languages defined by Jensen and Wirth and the IEEE Pascal Standards Committee draft.

Not all extensions can be translated to the base language efficiently. Even among the ones that can be translated adequately, there are some that cannot be handled by the proposed extension processor. The "structured constants" extensions are among the more popular ones that pose a difficult problem. Some special cases, such as single level arrays of simple base types, can be handled fairly adequately with the help of a few more sysfunctions and at least one additional construct in the TRANSLATE statement. The question of what minimal additions will be needed to handle the general case is currently under investigation by the author.

REFERENCES


APPENDIX

'Syntax of GRAMMAR and TRANSLATE statements

```plaintext
<grammar>
  <nonterm>  ::=  <identifier>  |  <token>
  <substitution>  ::=  <token>  <inline nonterm>  <alternates>
  <token>  ::=  <terminal>  |  <nonterm>  <token>  <alternates>
  <alternate label>  ::=  <nonterm label>  <substitution>  ...

<nonterm label>
  ::=  <identifier>  |  <terminal>  ...<identifier>  ...
<terminal>  is any token other than the special symbols :-:, ->, =:, · (, ), ,:: and ;;
<token>  ::=  <terminal>  |  <nonterm>
  <alternates>  ::=  <alternate label>  <alternates>  ...
  <nonterm label>  ::=  <nonterm>  <substitution>
```

From the collection of the Computer History Museum (www.computerhistory.org)