EXPRESSING PROGRAMMING LANGUAGE SYNTAX USING TEMPLATES

Jeffrey A. Brumfield

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-11 June 1985

Abstract

This paper proposes both a philosophy and a technique for specifying the syntax of programming languages. Our approach is oriented toward the programmer and is intended to compliment traditional methods such as Backus-Naur Form. Each language construct is represented by a template, which is a visual presentation of how the construct will appear in a program. The details of a language included in the templates may differ from those specified in a BNF description.

1. INTRODUCTION

A clear description of the syntax of a programming language is an important tool for anyone working with the language. Natural language descriptions of syntax can be so lengthy and complex that their accuracy may be questionable. For this reason, formal syntactic notations such as Backus-Naur Form (BNF) were developed. BNF or one of its dialects has been used to define the syntax of most major programming languages since Algol-60.

BNF has proved to be a concise way for a language designer to communicate syntax specifications to other language designers and to compiler implementors. BNF descriptions have also been provided to programmers as an absolute definition of syntax. Unfortunately, many programmers find BNF difficult and time consuming to read.

Syntax charts were developed in the early 1970's as an alternative to BNF. Syntax charts use diagrams to eliminate much of the special notation in BNF. Although less compact than BNF, syntax charts are more easily understood. For this reason, they have become the standard method for presenting language syntax to programmers.

To answer questions about syntax, many programmers prefer to study examples first, and to consult syntax diagrams only as a last resort. This suggests that syntax charts do not convey information in the most comprehensible way. Surprisingly, few efforts have been made in the last ten years to improve or replace syntax diagrams.

This paper suggests that, with respect to specifications of syntax, the needs of the programmer are different than those of the language designer and compiler implementor. A new technique, called templates, for describing the syntax of programming languages is presented. Templates can be used instead of syntax diagrams to communicate language syntax to programmers; templates are not intended to compete with BNF in terms of conciseness and compactness. The following section describes templates. Sections 3 and 4 discuss the design and organization of templates.

The details of a programming language that are included in its syntax specifications are limited by the descriptive capabilities of the notation being used. BNF, syntax diagrams, and templates all have the same descriptive power. However, to make templates more meaningful to programmers, we propose that certain details be omitted from templates and that other details not usually included in syntax specifications be suggested by the templates. Section 5 presents these ideas in detail.

Throughout the paper, we illustrate our ideas using examples from the Modula-2 language. Modula-2 has a greater variety of features than Pascal and consequently a more interesting syntax description. Because Modula-2 is simpler than languages such as Ada, we can present enough examples that the reader should feel confident that he could complete the syntax description.

2. TEMPLATES

A template describes a construct in a programming language. Figure 1 shows a template for a program module in the Modula-2 language. A template consists of keywords and symbols that are part of the language, references to other templates, and indicators of optional and repeated items. A template may describe an infinite set of things, such as all syntactically valid if statements, or a finite set of things, such as all binary operators.

Every template has a name, which is used to reference the template. To form a syntactically valid construct, each template name appearing within a template must be replaced by a sequence of characters that is described by the named template. Keywords must be distinguishable from template names. All Modula-2 keywords are in capital letters, so template names can be in lower case letters. In languages without this restriction, two different fonts can be used. For example, keywords can be boldface and template names can be in Roman, or keywords can be in Roman and template names can be in italics.

Within a template, items that are optional appear in shaded rectangles. Normally, if several items are enclosed within the same rectangle, then all of the items must be present or they all must be omitted. This rule can be overridden by nesting a more darkly shaded rectangle within the rectangle. If one shaded rectangle is nested within another, then the items within the inner rectangle are optional even if items in the outer rectangle are present. To indicate that the items within a shaded rectangle may be

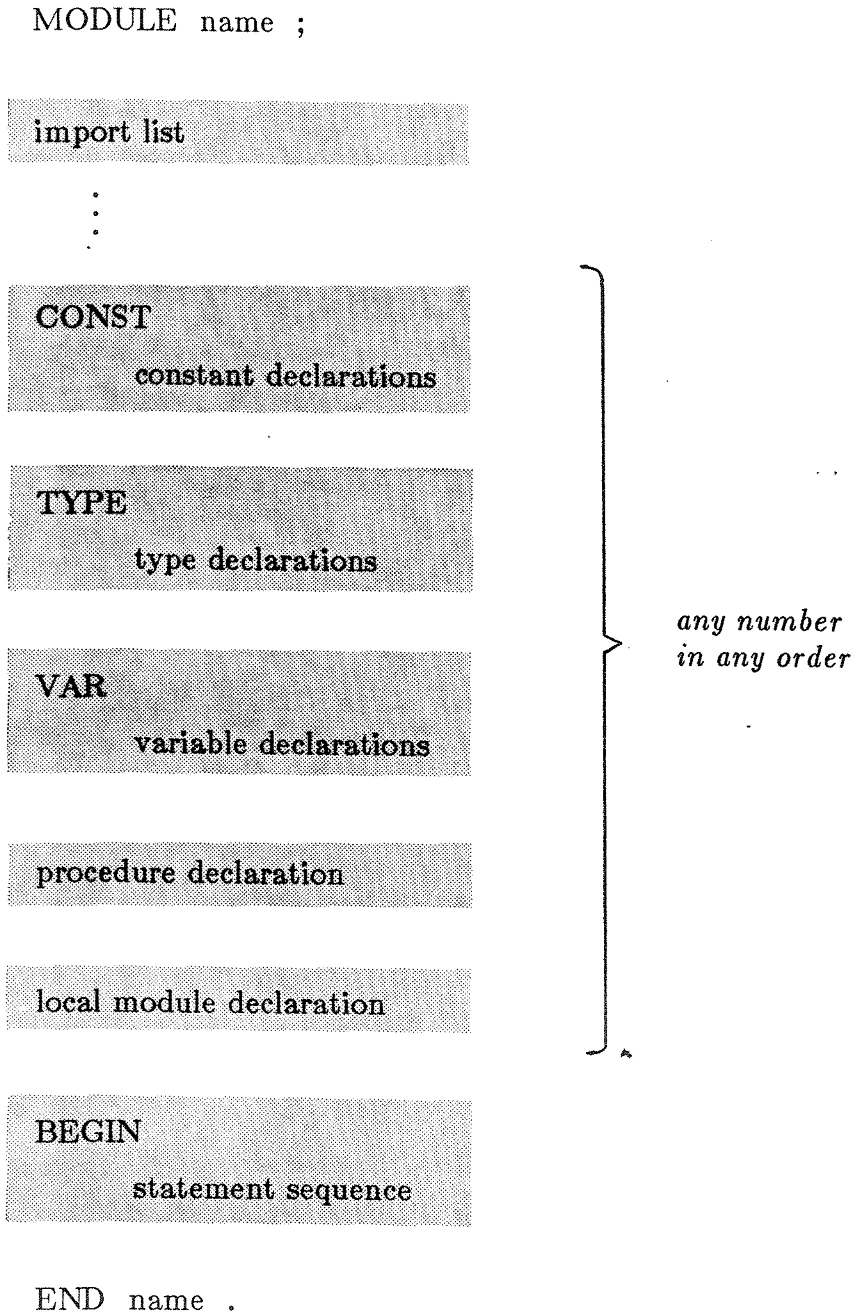


Figure 1. A template for a Modula-2 program module.

repeated any number of times, an ellipsis ("...") is placed immediately after or below the rectangle.

The template describing real numbers in Modula-2 illustrates optional and repeated items:

digit digit ... . digit ... E sign digit digit ...

It is apparent that a real number must contain at least one digit followed by a decimal point. If the number is written using scientific notation, then the letter "E" and at least one digit are required.

Shading alone cannot conveniently describe language constructs that have several distinct forms. One example of such a construct is the executable statement, whose variants may include an assignment statement, repetitive statements, and conditional statements. The syntax of each of these variants is totally different from the others.

A template for such a construct consists of a large opening brace ("{") that encloses the variants. If possible each variant is described on a single line. Thus, a Modula-2 statement is described by the template:

- assignment statement
- procedure call
- return statement
- if statement
- case statement
- while statement
- repeat statement
- for statement
- loop statement
- exit statement
- with statement

If multiple lines are needed for variants, then the variants can be separated by space or by a broken line.

When there are a large number of variants, each of which is a single symbol or keyword, the template may be presented more compactly without sacrificing clarity. The template for the Modula-2 binary operator could include an explanation in italics:

one of the following words or symbols:

```
+     -     *     /     DIV     MOD
AND     &     OR
=     <>     #     <=     >=     <     >     IN
```

In addition to specifying the syntax of constructs, templates also suggest proper coding style. For example, the template for an if statement

```
IF   boolean expression   THEN
         statement sequence

ELSIF   boolean expression   THEN
         statement sequence
   .
   .
   .
ELSE
         statement sequence

END
```

suggests that the keyword THEN should appear on the same line as the IF or ELSIF, and that the statement sequences should be indented.

To suggest that each statement should begin on a new line, the template for a statement sequence appears as

```
statement ;
statement
   .
   .
   .
```

instead of

```
statement  ;  statement  . . .
```

Since many entities have more than one acceptable format, the templates necessarily reflect the preference of their designer. There are certainly many stylistic issues that are not addressed by templates. Examples include the ordering of declarations and the

choice of names. Additional coding guidelines can optionally be listed with each template.

3. DESIGNING TEMPLATES

There are obviously many collections of templates for describing a given language. Selecting the most useful set of templates is more of an art than a science. This section presents several guidelines that may help in developing templates that will be meaningful to the programmer.

Templates should correspond to the components of a program in terms of which a programmer thinks. It is natural to have a template for each type of program unit, declaration, statement, and literal.

Repetition of details within two or more templates is frequently necessary for each template to convey a complete idea. A template should never be created solely to avoid repetition. For example, in the Modula-2 language main modules, implementation modules, local modules, and procedures may include the same types of declarations (i.e., constants, types, variables, procedures, and local modules). However, these different types of declarations are shown in each of the templates for completeness.

The different parts of a template should usually be described at the same level of detail. In particular, if a construct has several variants, each variant should include the same amount of detail. Consider the template for a statement presented in the previous

section. While the assignment statement, procedure call, return statement, and exit statement could easily be described in more detail on a single line, descriptions of the other statements are more complex. Consequently, each type of statement is described by a separate template. An additional benefit of doing this is that a descriptive name is associated with each statement type.

Too much information should not be presented in a single template. Unfortunately, there is not simple rule for deciding when a template has become too complex. Nesting shaded rectangles to show optional items should be used conservatively; more than two levels of shading is confusing and should be avoided.

Finally, the number of templates in the description of a language should be a consequence of following the above guidelines. Placing an arbitrary limit on the number of templates may lead to a poor design.

4. ORGANIZATION OF TEMPLATES

Once the individual templates for a language have been designed, they must be organized in a way that encourages their use. This task is complicated by the diverse needs of programmers. One programmer may want to casually examine the templates for all executable statements; another may want to study in depth the syntax of a single statement. To allow the templates to be accessed in several ways, we propose:

1) using a top-down organization,

2) grouping similar templates,

3) providing a table of contents, and

4) adding cross referencing information to the templates.

A top-down organization requires that the first templates describe a program (or whatever the units of compilation might be). Templates describing lower level constructs such as names and literals appear last. The top-down approach gives the novice some idea of the structure of a complete program; it can also motivate and organize an examination of the other templates.

Similar templates should be grouped for easy access. Natural categories for templates include data types, statements, expressions, and literals. Appendix A shows one way to group templates for the Modula-2 language. This list also serves as a table of contents that helps the programmer to quickly locate a specific template. The more complex the language, the more essential the table of contents is.

There may be a few templates that do not seem to belong to any category. Frequently these templates are referenced only once and can be placed near the templates that reference them.

Each template can be assigned a number of the form $x.y$, where x specifies the group to which the template belongs and y is the number of the template within the group. To facilitate locating other templates referenced within a template, their numbers can appear in the right margin on the same line as the reference. If more than

one template is referenced on a line, the numbers appear in the same order as the template names. If a template is referenced more than once on a line, only the first reference needs to have an entry in the list of numbers. Appendix B gives several templates that use this cross referencing scheme.

5. THE CONTENT OF SYNTACTIC DESCRIPTIONS

Formal syntactic notations have the ability to specify only certain details of a programming language. For example, the notations described in this paper are powerful enough to specify that parentheses in an expression must be balanced or that there must be a corresponding END for each procedure heading. However, these notations are not powerful enough to specify that a procedure call must have the same number of parameters as the procedure declaration or that the name following the procedure END must be the same as the name in the procedure heading.

Language designers usually formulate their syntax descriptions to help the compiler implementor check for as many errors as possible during the parsing of a program. Some details included in these descriptions are needlessly confusing to the programmer. Other details usually omitted can convey important information to the programmer. This section gives two examples of how the information in templates might differ from that presented in traditional syntax descriptions.

One of the most confusing parts of the syntactic specification of many programming languages is the description of an expression. The following are the Extended BNF rules describing Modula-2 expressions:

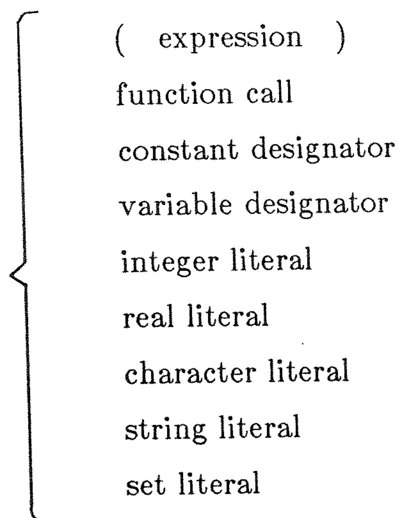
```
expression        = SimpleExpression [ relation SimpleExpression ] .
SimpleExpression  = [ "+" | "-" ] term { AddOperator term } .
term              = factor { MulOperator factor } .
factor            = number | string | set | designator [ ActualParameters ] |
                    "(" expression ")" | NOT factor .
relation          = "#" | "<>" | "=" | "<" | "<=" | ">" | ">=" | IN .
AddOperator       = "+" | "-" | OR .
MulOperator       = "*" | "/" | DIV | MOD | AND | "&" .
```

The complexity of these rules results from the desire to indicate operator precedence and thereby make the specifications unambiguous. This is important to the compiler implementor in guaranteeing a unique parse of every expression.

While operator precedence is also an important concern of the programmer, it can easily be omitted from the syntactic specifications and described with the semantics. The resulting templates convey the true simplicity of an expression:

unary operator operand binary operator operand . . .

The unary operators are +, -, and NOT; the binary operators were listed in Section 2. An operand is specified by the template



Most programming languages use names (or identifiers) to reference objects such as variables, types, and procedures. The syntax of names for all types of objects is usually the same. For this reason, we often see the term *name* used throughout a syntactic description with no indication of what objects the name can refer to.

Templates can be made more meaningful to the programmer if additional information is specified for names. For example, the template for a Modula-2 import list

FROM module name IMPORT name , name . . . ;

shows that the name following the keyword FROM must be that of a module, but the list of names can include names of any objects.

To avoid creating several identical templates, we allow a single template to have several names. Thus, *name, variable name, type name, procedure name,* and *module name* all reference the same template. There are two instances when we do not specify

the objects to which names refer. The first is when an object is being declared; the second is when any object name may appear.

Data type compatibility is also not usually shown in syntactic specifications. Except for the set of operators that is allowed, the syntax of all expressions is the same. Nevertheless, when an expression must be of a certain type we show that fact in the template name. For example, the terminating condition for while and repeat statements is referred to as *boolean expression* instead of simply *expression*.

6. DISCUSSION

Templates are as powerful and precise a notation for specifying syntax as BNF or syntax diagrams. Unlike BNF, templates require no meta-notation that could be confused with symbols in the language. The meta-notation used in templates consists of shading, font changes, the large brace, and the ellipsis. Unlike syntax diagrams, templates show how constructs will appear in a program. In this way a template has many advantages of an example, while still being as general as possible.

The use of shading makes the simplest form of each language construct easy to discern. More complex forms are constructed by including items in shaded rectangles and possibly repeating them.

While a template may show several variants of a construct, we have not proposed notation that can be used to indicate alternatives within a template. The absence of this capability has not proved to be an inconvenience. Although the resulting templates are less compact, they are visually more meaningful.

Template names have not been restricted to single words. Therefore, two consecutive template names must be separated by shading or white space. If the template designer feels this can lead to confusion, template names can be hyphenated or changed to compound words.

Templates could be used to convey the same information as BNF descriptions. However, we believe the creation of syntax diagrams directly from BNF rules has prevented them from being more useful to programmers. While BNF is still ideal for language designers and compiler implementors, templates can convey slightly different information to meet the needs of programmers.

The ideas proposed in this paper are intended to be flexible. We have used them to describe the syntax of a few Pascal-like languages. The template designer is encouraged to modify and extend our techniques if doing so will allow other languages to be described clearly and simply.

APPENDIX A

Organization of Templates for the Modula-2 Language

Compilation Units

1.1 program module
1.2 definition module
1.3 procedure heading
1.4 implementation module

Declarations

2.1 import list
2.2 export list
2.3 constant declarations
2.4 type declarations
2.5 definition module type declarations
2.6 variable declarations
2.7 procedure declaration
2.8 formal parameter list
2.9 formal parameters
2.10 local module declaration

Data Types

3.1 type
3.2 enumerated type
3.3 subrange type
3.4 array type
3.5 index type
3.6 record type
3.7 field list sequence
3.8 field list
3.9 variant list
3.10 set type
3.11 base type
3.12 pointer type
3.13 procedure type
3.14 formal type list
3.15 formal type

Statements

4.1 statement sequence
4.2 statement
4.3 assignment statement

4.4 procedure call
4.5 return statement
4.6 if statement
4.7 case statement
4.8 selector list
4.9 selector
4.10 while statement
4.11 repeat statement
4.12 for statement
4.13 loop statement
4.14 exit statement
4.15 with statement

Expressions

5.1 expression list
5.2 expression, boolean expression
5.3 unary operator
5.4 binary operator
5.5 operand
5.6 function call
5.7 constant expression
5.8 constant operand

Names

6.1 name list
6.2 name, module name, type name, field name
6.3 constant designator, type designator
6.4 variable designator, procedure designator
6.5 qualifier
6.6 alphanumeric

Literals

7.1 integer literal
7.2 real literal
7.3 character literal
7.4 string literal
7.5 set literal
7.6 element
7.7 digit
7.8 octal digit
7.9 hex digit
7.10 sign

APPENDIX B

Sample Templates for the Modula-2 Language

2.7 procedure declaration

PROCEDURE name (formal parameter list) : type designator ; [6.2, 2.8, 6.3]

CONST constant declarations	*any number in any order*	[2.3]
TYPE type declarations		[2.4]
VAR variable declarations		[2.6]
procedure declaration		[2.7]
local module declaration		[2.10]

BEGIN statement sequence [4.1]

END name ; [6.2]

2.8 formal parameter list

formal parameters ; formal parameters . . . [2.9]

2.9 formal parameters

VAR name list : ARRAY OF type designator [6.1, 6.3]

3.6 record type

RECORD	
field list sequence	[3.7]
END	

3.7 field list sequence

field list ;	[3.8]
field list	[3.8]
⋮	

3.8 field list

{ name list : type	[6.1, 3.1]
{ variant list	[3.9]

3.9 variant list

CASE **name :** type designator OF	[6.2, 6.3]
selector list : field list sequence |	[4.8, 3.7]
selector list : field list sequence	[4.8, 3.7]
⋮	
ELSE field list sequence	[3.7]
END	

4.10 while statement

```
WHILE  boolean expression  DO                                   [5.2]
        statement sequence                                      [4.1]
END
```

4.11 repeat statement

```
REPEAT
        statement sequence                                      [4.1]
UNTIL  boolean expression                                       [5.2]
```

4.12 for statement

```
FOR variable designator := expression TO expression BY constant expression DO
                                                       [6.4, 5.2, 5.7]
        statement sequence                                      [4.1]
END
```

4.13 loop statement

```
LOOP
        statement sequence                                      [4.1]
END
```

4.14 exit statement

```
EXIT
```

4.15 with statement

```
WITH  variable designator  DO                                   [6.4]
        statement sequence                                      [4.1]
END
```